Java 动态类加载分析*)

左天军[1] 朱智林[1] 韩俊刚[2] 陈 平[1]
(西安电子科技大学软件工程研究所 西安710071)[1] (西安邮电学院计算机系 西安710061)[2]

摘 要 动态类加载是Java的一个重要功能,它支持Java在运行时安装程序组件。Java的动态加载具有惰性加载、用户自定义加载策略以及动态名字空间等新特征。本文详细讨论了Java的动态加载机制,研究了动态加载与Java平台安全性之间的关系,分析了针对类加载的典型攻击,讨论了形式化验证的方法和存在的相应问题,最后总结全文并指出进一步研究的方向。

关键词 动态加载,Java安全性,动态连接,形式化方法

The Analysis of Java Dynamic Class Loading

ZUO Tian-Jun[1] ZHU Zhi-Lin[1] HAN Jun-Gang[2] CHEN Ping[1]
(Institute of Software Engineering, Xidian University, Xi'an 710071)[1]
(Department of Computer, Xi'an Institute of Posts and Telecommunications, Xi'an 710061)[2]

Abstract Dynamic class loading is a novel and powerful mechanism of Java. It supports installing software components at runtime. Java allows lazy, dynamic loading of classes according to user-definable policies, and a form of name space separation using class loaders. In the paper, we present the notion of Java class loading in detail. We show how class loading is closely related to the security of Java platform and analyze typical attacks on loading. We also present formalizations of class loading and show the existing problems of these works. We propose the future research at last.

Keywords Dynamic class loading, Java security, Dynamic linking, Formalization

1 引言

动态加载是一种在运行时安装程序组件的技术。许多操作系统如Multics[1], SunOS 4.0[2]以及Microsoft Windows使用的动态连接就是一种动态加载技术。使用动态连接后,程序中的符号引用可以在程序被加载到内存后才替换成相应的机器地址。并且,在多数实际系统中动态连接是惰性的,即直到第一次使用时符号引用才被替换。被动态加载到内存的软件组件可以同时被多个进程共享,并且在硬盘上只需保留一份拷贝,因此与静态连接相比,动态连接一个主要优点是可以节约磁盘和内存空间。动态连接的另一个优点是增加了程序设计的灵活性。当程序使用的动态连接代码需要更新时,不需要重新编译和连接该程序。当然,采用动态连接的程序在启动时会稍慢于静态连接的程序,不过对于现代的CPU来说这通常不是问题。在有些程序设计语言Common Lisp[3]中也提供了动态加载和连接程序的功能。

与上述系统相比,Java的动态加载技术具有一些新特点:(1)用户自定义加载策略。例如应用程序可以根据需要从网络或本地文件系统中加载代码;(2)动态名字空间,即程序被划分成相互独立的运行空间,名字相同的代码可以动态地并存于同一个程序中。

动态加载是Java平台的一种重要的底层实现机制,它与程序设计的灵活性和可靠性紧密相关,深入了解这种技术对于开发Java应用程序是非常必要的。为此,本文从应用程序开发、安全性、形式化验证等方面详细讨论了Java的动态加载技术。本文第2节介绍Java平台的动态加载机制,包括用户自定义加载策略、定义加载器和初始化加载器等概念;第3节详细分析Java动态加载技术的安全性问题;第4节讨论动态加载的各种形式化验证技术及其存在的问题;最后总结全文并指出进一步研究的方向。

2 Java平台的动态加载机制

在标准实现中,Java编译器为Java程序的每个类生成一个类文件,这种文件采用Java虚拟机(JVM)定义的类文件格式保存,由JVM加载并解释执行。在JVM规范[4]中,加载和连接被定义成两个相互递归的过程。加载指JVM根据指定的类名或接口名寻找相应的类文件,并将类文件安装到运行环境中的过程。连接指JVM对类文件进行验证和解析符号引用的过程。图1给出了加载和连接与Java编译器、JVM之间的关系。

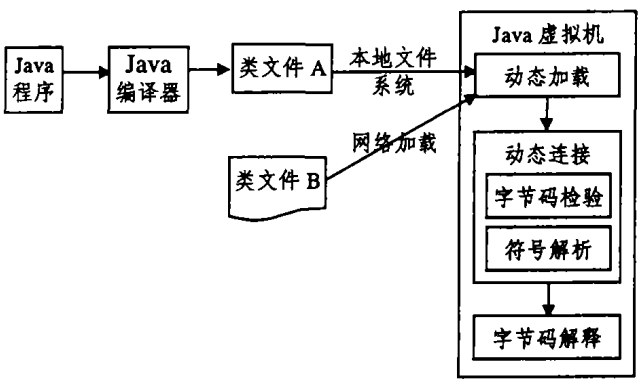

图1 类加载和连接与Java编译器、JVM的关系

JVM的动态加载是通过Java系统类java.lang.ClassLoader或其子类实现的。java.lang.ClassLoader是一个抽象

*)Supported by the Natioal Natural Science Foundation of China under Grant No. 90207015(国家自然科学基金).左天军 博士,主要研究领域为网络安全、形式化验证、软件工程技术;朱智林 博士;韩俊刚 教授,博士生导师;陈 平 教授,博士生导师。

类，其实例或其子类的实例称为类加载器。指定一个类名后，类加载器会寻找相应的类文件并将其安装到JVM环境中，同时为该类创建一个相应的Class对象。通常，JVM从本地文件系统中加载类文件。例如，在UNIX系统中，JVM从CLASSPATH环境变量定义的目录中加载类。同时，JVM也允许应用程序通过实现ClassLoader的子类定义其加载策略。例如，图2给出了一个从网络加载类文件的实例。

```
class NetworkClassLoader extends ClassLoader {
        private String netAddress;
        private int port;
        public NetworkClassLoader ( String netAddr, int
                pt){
                netAddress=netAddr;
                port=pt;
        }
        public Class loadClass(String className){
              byte [ ]data = getClassData (netAddress,
              port, className);
              return defineClass(className, data, 0, data-
              .length);
        }
}
```

图2 用户定义的加载策略

图2中的loadClass是ClassLoader类中的一个方法，它根据指定的类名加载类文件并返回一个保存该类运行时信息的Class对象，应用程序可以通过重置这个方法定义其加载策略。通常loadClass加载类文件时，先根据类文件结构创建一个代表该类的字节数组，然后通过defineClass从这个字节数组中导出相应的类并返回一个Class对象。defineClass是ClassLoader类中的一个final方法，不能在应用程序的子类中重置。

定义1(初始化加载器) 如果采用L.loadClass()加载一个类C并返回一个Class对象，则称L为类C的初始化加载器，或称L初始化类C的加载。

定义2(定义加载器) 如果采用L.defineClass()加载一个类C并返回一个Class对象，则称L为类C的定义加载器，或称L定义类C。

在Java应用程序中，类的加载是可以委托的，一个加载器可以请求其它加载器为其加载类文件。因此，在委托模式中一个类的初始化加载器与定义加载器可以不必相同。

在Java平台中，所有类是由类加载器加载的，也就是说一个加载器的类也是由其它类加载器加载的。为了解决第一个加载器是如何被加载的问题，Java采用一个特殊的加载器——主加载器来完成所有程序的最初加载，这个加载器通常采用一种本地语言(如C)来实现，以一种平台相关的方式从本地文件系统中加载类文件。在Java平台中，主加载器用NULL表示。在JDK 1.2中，类加载器的类层次关系≤可以表示为：主加载器≤java.lang.ClassLoader≤java.security.SecurityClassLoader≤java.net.URLClassLoader≤sun.Applet.AppletClassLoader。

定义加载器为类定义了动态名字空间。一个类在编译时，其类型是由类名静态确定的；一个类在运行时，其类型是由类名和类的定义加载器共同确定的。动态名字空间的引入使得Java程序的类型检查不能仅仅在编译时完成，并且为Java的安全性带来了问题。

3 安全性分析

动态加载不仅是Java语言的核心机制，而且与Java平台的安全模型紧密相关。最初，Java安全架构采用一种"沙箱"模型。在这个模型中，所有从网络获得的远程代码，即小程序(applet)，被认为是不可信代码；所有本地代码被认为是可信代码。本地代码可以访问所有重要的系统资源，如文件系统；远程代码仅能访问非常有限的系统资源。由于JDK 1.0支持这种沙箱模型，因此通过JDK开发的大多数应用程序(如支持Java的Web浏览器)实现的也是这种安全访问机制。为了扩展这种机制，JDK 1.1提供了一种签名机制，凡是具有正确数字签名的小程序也被当成可信的代码。JDK 1.2实现了一种访问粒度更细的安全模型，Java系统可以通过安全策略控制应用程序的访问权限，代码的可信度是由其访问权限确定的。在这种模型中，本地代码不一定是完全可信的，同远程代码一样，它也需要根据其拥有的权限访问系统资源。类加载器在JDK 1.2的安全模型中起着重要的作用，因为它关系到类文件的定位和获取以及如何根据适当的权限定义相应的类对象。为了确保安全性，Java严格定义了每个加载器所能加载的类文件以及应用程序和小程序所能创建的加载器。例如，Java仅允许由主加载器加载系统类；应用程序和小程序只能通过JDK 1.2提供的静态方法创建URLClassLoader类的实例。

围绕类加载的攻击有很多种方式，其中比较典型的有针对SecurityManager的攻击和类型欺骗。Java系统类SecurityManager用于检查应用程序的访问权限，Java的安全模型要求所有Java 2 SDK系统代码必须调用SecurityManager进行安全检查。如果一个程序加载一个新的SecurityManager时修改了其中的某些变量的属性，例如，在AppletSecurity类中，变量initACL用于跟踪ACL是否被初始化；变量networkMode用于确定小程序所能进行的网络连接，如果程序将networkMode设置为可以进行任何网络连接，initACL设置为true，那么就可以绕过Java的所有安全检查。

另外，SecurityManager通常采用栈检测技术动态确定一个调用是否由某个远程代码发起。由于主加载器在Java平台中采用NULL表示，因此如果一个类加载器在创建一个Class对象时将该对象的定义加载器属性设置为NULL，那么就可以让Java运行系统相信这是一个本地代码，由此绕开SecurityManager的安全检查。

类型欺骗是Java早期版本(JDK 1.0和JDK 1.1)中的一个典型的安全漏洞。考虑图3中的程序。本文采用$(N,L_1)^{L2}$表示类名为N、定义加载器为L_1、初始化加载器为L_2的类。当不需要考虑初始化加载器时写为(N,L_1)；当不需要考虑定义加载器时写为N^{L2}。

```
class (RT,L1){
    private R r;
    void test(){
        RR rr=new RR();
        r=rr.getR();//Fail
        r.k=r.k+1;
    }
}
class (R,L1){
    public int k;
}
```

```
class (RR,L2){
    R getR(){
        return new R
        ();
    }
}

class (R,L2){
    private object k;
}
```

图3 类型欺骗问题

为了理解类型欺骗，必须先了解Java的类解析算法。Java系统通常采用如下步骤解析类：(1)检查该类是否已被加载，如果是，就没有必要再次加载；(2)选定加载该类的加载

器。由于系统中存在着多个加载器，Java 采用以下规则选定一个类的加载器：(1)采用一个新的 URLClassLoader 对象实例加载应用程序的第一个类；(2)采用一个新的 AppletClassLoader 对象实例加载小程序的第一个类；(3)如果在当前类中引用了一个类名或通过 java.lang.Class.ForName ()引用了一个类名，那么采用当前类的定义加载器加载所引用的类。

现在分析图3中类型欺骗出现的原因。当程序运行到〈RT，L_1〉的 RR rr=new RR()语句时，由于程序引用了类 RR，Java 采用类 RT 的定义加载器 L_1 加载 RR，但是在加载过程中 L_1 将加载委托给 L_2。当程序运行到〈RT，L_1〉的 r=$rr.getR$()语句时，由于引用了类 R 的字段 r，Java 通过字段解析算法为 r 加载类〈R，L_1〉；由于引用了 $getR$()，Java 通过方法解析算法在〈RR，L_2〉中查找并调用方法 $getR$()。当程序运行到〈RR，L_2〉中的 return new R()语句时，由于引用了类 R，Java 采用类 RR 的定义加载器 L_2 加载并创建一个类〈R，L_2〉的对象。于是，程序在语句 r=$rr.getR$()处出现了类型欺骗，r 的类型是〈R，L_1〉，但 $rr.getR$()返回的类型却是〈R，L_2〉。因此，通过 r=$rr.getR$()，r 实际指向的是〈R，L_2〉中的属性 private object k。这样，当程序运行到 $r.k$=$r.k$+1时，程序不仅可以访问〈R，L_2〉中的私有属性 k，而且可以将 object 指针伪造成 int 类型进行代数运算，从根本上突破了 Java 的类型系统。

为了解决类型欺骗问题，JDK 1.2在 JVM 内部维护了两个数据结构并确保这两个数据结构是一致的，这两个数据结构是被加载类高速缓存和类加载约束集合。被加载类高速缓存是一个从类名和初始化加载器到相应类型的映射；类加载约束集合采用以下规则定义：(1)属性引用规则。如果〈C，L_1〉引用了〈D，L_2〉中的一个静态类型为 T 的属性，那么就在集合中增加约束 $T^{L1}=T^{L2}$；(2)方法引用规则。如果〈C，L_1〉引用了〈D，L_2〉中的方法 T_0 method (T_1，…，T_n)，那么增加约束 $T_0^{L1}=T_0^{L2}$，$T_1^{L1}=T_1^{L2}$，…，$T_n^{L1}=T_n^{L2}$；(3)方法重置规则。如果〈C，L_1〉重置了〈D，L_2〉中的一个 T_0 method (T_1，…，T_n)方法，那么增加约束 $T_0^{L1}=T_0^{L2}$，$T_1^{L1}=T_1^{L2}$，…，$T_n^{L1}=T_n^{L2}$。

Java 规定如果以下条件不能同时成立，那么被加载类高速缓存和类加载约束集合是一致的：(a)存在一个加载器 L，并且 L 被 JVM 记录为以 N 为类名的类 C 的初始化加载器；(b)存在一个加载器 L'，并且 L' 被 JVM 记录为以 N 为类名的类 C' 的初始化加载器；(c)存在类加载约束 $N^L=N^{L'}$；(d)$C\neq C'$。

为了维护以上一致性约束，每加载一个类时，Java 就检查这个类是否不违反所有约束条件，若违反其中任意一个约束则加载失败；每增加一个新的约束条件时，Java 就检查此约束是否能满足所有已加载的类，若不能满足则约束增加失败。

Saraswat[5]最早提出类型欺骗问题并给出两种解决方案。一种是在运行时进行类型检查；另一种与 JDK 1.2的类加载约束非常相似。文[5]与 JDK 1.2最大的不同在于方法重置的处理上，文[5]认为动态类型(类名和类加载器)相同才能方法重置，JDK 1.2认为只要静态类型(类名)相同就可以重置。

4 形式化验证及其问题

由于 Java 动态加载的复杂性，采用形式化方法描述和验证其正确性是非常必要的。Dean[6]在 PVS 系统中研究了动态连接和静态类型检查之间的关系并给出了一个与 Java 动态连接非常近似的模型。为保证动态连接相对于静态类型检查是正确的，文[6]要求模型中的类是单调递增的，这个约束与 JDK 1.2要求的对于同一个类名只能用同一个加载器加载一次的约束相类似。Jensen[7]描述了 JVM 动态加载和连接的操作语义，并分析了类型欺骗问题出现的原因。但是，Jessen 没有描述动态加载和连接的内部实现。并且，与 JVM 规范相比 Jessen 的方法存在着不准确之处。例如，文[7]错误假设在类的层次结构中，所有类是由同一个加载器定义的。实际上 Java 规范要求：如果程序中的当前类引用了另一个类，那么就用当前类的定义加载器加载那个类。Tozawa[8]提出一个描述动态加载的操作语义模型，Tozawa 采用环境的概念定义了 invokevitual 和 areturn 指令的状态转移关系。Fong[9]认为动态加载、连接和字节码检验之间的耦合过于紧密，为此提出一种模块结构研究此问题，文[9]形式化了所提出的模块结构并给出正确实现此结构需要满足的一致性约束条件。但是，文[9]没有考虑多个加载器的情况，因此文[9]的类加载模型要比 Java 规范简单。Drossopoulou[10]提出了一个用于 Java 连接和检验的抽象模型，Drossopoulou[11]提出了一个描述动态连接的非确定模型，它能够用统一的方法研究 Java 和 C# 两种不同的动态连接模型。但是，文[10,11]没有考虑动态加载的实现细节以及多加载器的情况。

总的来说，以上方法存在的共同问题是：(1)多加载器是 Java 规范中的核心概念，但是以上模型没有考虑多加载器的具体实现；(2)Java 的类型检查分布于编译、动态加载、动态连接和运行四个阶段，这几个阶段是相互联系和影响的。但是以上方法在研究时仅仅考虑了其中某几个阶段，这样就不足以确保整个 Java 平台的正确性。

结论 从以上分析可以看出，Java 的动态加载在程序设计中具有重要的地位，一方面通过可定义的加载策略和动态名字空间为程序设计提供了极大的灵活性，另一方面也让 Java 的类型系统变得更加复杂并由此带来设计和实现上的错误。因此，深入了解动态加载技术对于 Java 应用程序的开发具有重要的意义。由于动态加载的复杂性，采用形式化方法研究这个问题是非常必要的，进一步的研究主要是采用统一的模型描述和验证 Java 编译、加载、连接和运行时的类型系统。

参考文献

1 Organick E. The Multics System: An Examination of its Structure. MIT Press, Cambridge, Massachusetts, 1972

2 Gingell R A, Lee M, Dang X T, Weeks M S. Shared libraries in SunOS. In: USENIX Conf. Proc. Phoenix, AZ, 1987. 131～145

3 Keene S E. Object-Oriented Programming in Common Lisp. Addison-Wesley, 1989

4 Lindholm T, Yellin F. The Java™ Virtual Machine Specification (2nd edition). Addison-wesley, 1999

5 Saraswat V. Java is not type-safe: [Tech. Rep]. AT&T Research, Florham Park, New Jerse, Aug. 1997

6 Dean D. The security of static typing with dynamic linking. In 4th Computer and Communications Security (Zurich, Apr.), ACM, New York, 1997. 18～27

7 Jensen T, Metayer D L, Thorn T. Security and dynamic class loading in Java: A formalization. In Computer Languages, IEEE Comput. Soc. Press, Los Alamitos, Calif., 1998. 4～15

8 Tozawa A, Hagiya M. New fomalizaion of the JVM. http://nicosia.is.s.u.-tokyo.ac.jp/members/miles papers/cl-99.ps, 1999

9 Drossopoulou S. Towards an abstract model of Java dynamic linking and verification. In: Harper R, ed. TIC'00 - Third Workshop on Types in Compilation (Selected Papers), volume 2071 of Lecture Notes in Computer Science, Springer, 2001. 53～84

10 Drossopoulou S, Lagorio G, Eisenbach S. Flexible models for dynamic linking. In: Degano P, ed. European Symposium on Programming 2003, volume 2618 of Lecture Notes in Computer Science, Springer, 2003. 38～53

11 Fong P W L, Cameron R D. Proof Linking: Modular Verification of Mobile Programs in the Presence of Lazy, Dynamic Linking. ACM Transactions on Software Engineering and Methodology, 2000, 9(4): 379～409