|  |  |  |  |
| --- | --- | --- | --- |
|   |

|  |  |
| --- | --- |
|  |  |

 |

|  |  |  |
| --- | --- | --- |
|  | Java 安全的演进这篇论文提供了 Java 安全发展和演进的高级概述。Java 是一项日趋成熟的技术，它从作为一种基于浏览器的脚本编制工具的商业起源演进而来。我们探讨了以 Java 为目标的不同部署环境、Java 运行时的一些特征、这种基本技术当前发行版中的安全功能、新的 Java Development Kit（JDK）1.2 的基于策略的安全模型、基于堆栈的授权安全模型的局限性、通用的安全要求以及 Java 安全将来可能的走向。IBM 关于 Java 安全的倡议考虑到了客户的部署基于 Java 的企业解决方案的愿望。因为写这篇论文时 JDK 1.2 正进入企业 beta 测试阶段，所以在 JDK 1.2 被广泛采用时，业界反馈可能会引起一些操作上的更改和改进。软件业致力于为开发和部署用 Java 编写的任务关键的应用程序提供支持。Java 环境涵盖了从企业服务器到嵌入式设备的广阔范围。在众多基于 Java 的系统中，包括 JavaOS、EmbeddedJava 和 PersonalJava 在内的系统将变得可用，潜在地提供不同级别的底层服务。这种局面将会导致对各种不同级别的安全强度的需求。Java 安全最初的焦点是对在万维网浏览器中所下载的 applet（小程序）的支持。Java 的安全功能很大程度上反应了对这种功能的继承。随着 Java 的成熟，它将日益增加对附加的安全功能的支持，以满足目标应用程序环境的需要。所包括的附加功能是在大规模服务器应用程序中常见的安全功能。我们已经看到过电子商务（通过 Web 电子化经营的商务）怎样通过大量的定单增加客户范围的示例。随着客户群的扩大，由于恶意行为所造成的绝对损失量可能大大增加，有必要在信息技术系统中部署改进的安全产品。但是，如果安全漏洞被广泛公开，客户的信誉将受到玷污。IBM 的客户要求系统实现几乎是完美无缺的，并且能够满足企业的需要。他们期望 IBM 确保风险是公开可知的，并且当漏洞被揭开时能够马上作出反应。Gartner Group 的 1996 年 12 月所做的报告，Java -- Good Start, but Not Yet Secure [1](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml#1)，强调了关于基于 Java Development Kit（JDK）早期版本的 Java 安全所涉及的几个方面。切合实际的期望是很重要的：因为没有系统是百分之百安全的。但是，认识到关于 Java 安全的这三方面是很关键的：1. Java 启用了以前从未被大范围商业采用的功能：动态地从系统之外的源文件装入代码。这个重要的功能使得著名的“特洛伊木马”安全问题更严重。但是，它也提供了极有价值的功能。
2. Java 安全并不是为了解决与 Operating System/390(OS/390) Security Sever 中的“资源存取控制设施”（Resource Access Control Facility（ RACF））功能相同的问题，这个问题通过“多虚拟存储”（Multiple Virtual Storage（MVS））或类似的传统企业安全技术得到解决。RACF 旨在保护企业及其资源免于受到恶意用户的攻击。而 Java 安全旨在保护用户的工作站和资源免于受到恶意代码的攻击。
3. RACF 和其它传统企业安全技术工作得很好，因为它们采用 — 并且它们的环境提供 — 强大的操作系统完整性作为基础。Java 也采用这种完整性；但是，主流的桌面操作系统没有充分提供这种完整性。

安全技术需要预防或减少以下这几种类型的威胁：* 未授权的资源使用，包括盗用软件或 CPU、破坏数据和软件、信息泄漏以及不负责任的行为
* 滥用特权，包括虚报标识符、从属关系、交换的项目值以及服务的授权，假冒，欺骗以及勒索
* 恶意的代码，包括需要防范的病毒、蠕虫、特洛伊木马以及逻辑炸弹
* 窃听，包括主动和被动手段
* 拒绝服务，包括资源毁坏、服务饱和以及通信中断

到目前为止，特洛伊木马和电子欺骗攻击是公布的最常见的 Java 安全威胁。因为大多数攻击是由于浏览器或 Java 中的实现错误，而不是基本的设计错误而发生，所以一旦确定，比较容易提供修复。Java 安全的基础及演进因为最初 Java 的商业部署是在 Web 浏览器中，所以 Java 安全的侧重点是提供保护程序不受到恶意 applet 攻击的功能；也就是说，保护不受到从因特网的 Web 站点下载的恶意代码的攻击。Java 安全建立在 Java 运行时环境的三个基本方面的基础上：ByteCode Verifier（字节码验证器）、Security Manager（安全管理器）以及 ClassLoader（类装入器）。ByteCode Verifier 确保所下载的代码被恰当地格式化，字节码（“Java 虚拟机”指令）没有违反这种语言或虚拟机的安全限制（无非法数据转换），没有执行指针寻址，内部堆栈不能溢出或下溢，以及字节码指令将拥有正确的类型参数。[2](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml%22%20%5Cl%20%222)Security Manager 在尝试执行文件 I/O 和网络 I/O、创建新的 ClassLoader、操作线程或线程组、启动底层平台（操作系统）上的进程、终止“Java 虚拟机”、将非 Java 库（本机代码）装入到 JVM、完成某种类型的窗口系统操作以及将某种类型的类装入到 JVM 中时发起运行时访问控制。例如，Java applet 沙箱 [3](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml#3) 严格地将所下载的 applet 约束到被认为是比较安全的有限的一组函数中。ClassLoader 确定 applet 怎样以及什么时候可以装入代码并确保 applet 不会取代运行时环境中的系统级组件。除此以外，Java 编程语言和运行时环境的一些功能，包括自动的内存管理和强大的数据类型安全，使编写安全的代码更容易。通过使用 JDK 1.1 提供的数字签名服务，可以以与用 Java 编写的应用程序类似的方式对待可信的 applet。也就是说，这些可信的 applet 比在被限制的 Java 沙箱中运行的 applet 对 JVM 资源具有更高的访问权。改进过的并且更加灵活的访问控制功能是 JDK 1.2 中添加的主要安全功能，将在这篇论文的后面的部分中更详细地描述。今天的计算环境有很多安全方面的弱点，早于 JDK 1.2 的 Java 中的可用功能克服了这些弱点。* 强大的内存保护 — Java 除去了恶意或不经意读取或毁坏程序边界外的内存位置的可能性。结果，Java 应用程序或 applet 不能获得读取或更改内容的未授予的内存访问权。
* 加密和数字签名 — Java 支持使用强大的加密技术来验证 applet 是否来自一个可以识别的来源，并且没有被修改过。
* 规则实施 — Java 是完全基于对象的。通过使用 Java 对象和类来表示共同的信息实体，有可能显式地声明管理这种对象使用的规则。

http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image1.gifJava 运行时环境Java，作为一种面向对象的语言，可以用来以与用 C 和 C++ 来提高程序员生产效率几乎相同的方式开发应用程序。与其它很多编程语言相比较，Java 提供了标准的一组库，包括范围广泛的通信和安全功能，因此简化了客户机／服务器以及分布式系统应用程序的构造和部署。当代码被装入到 JVM 运行时环境时，它还提供了数据类型安全，并且完成字节码验证。这捕获了由编程错误产生的错误，尤其是由指针算法和数组越界索引的错误。此外，基于 Java 的支持环境有下面这些：* Applet — 具有被限制访问权的可下载代码，通常和浏览器配合使用
* Aglet — 推向终端设备的代码，而不是 applet 的拉模型
* Servlet — 服务器上的 Java 代码
* Orblet — 使用对象请求代理通信机制的代码

为了达到可重用性，这些环境的任何一种都可以使用组件模型，比如 JavaBeans [4](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml#4)和 Enterprise JavaBeans [5](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml#5)。Java 正发展为支持大多数的系统配置：* 目标为在家庭以及其它地方的多种消费者设备中使用的嵌入式系统
* 基于 Java 的智能卡
* 像寻呼机和 PDA（个人数字助理）这样的个人管理设备
* 网络计算机 — 瘦客户机和胖客户机
* 移动系统
* 高可伸缩的企业服务器

为了支持这个范围的配置，正在发展一系列的产品 — 包括 JavaOS、PersonalJava、Enterprise Java 以及驻留在应用程序构建器环境中的 JVM — 来满足各个环境中独特的需要。安全方面的要求随这些实质上不同环境的独特特征的不同而异。最初的工作表明通用的安全模型是可能的，环境的差异可以通过提供扩展来支持，而不必部署截然不同的安全模型。通用安全模型的使用将简化和减少应用程序和库开发与部署的成本。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image2.gifJDK 1.2 权限模型下面的讨论基于 beta 级代码；在普遍可用之前，业界反馈可能会引起一些操作上的更改。JDK 1.2 引入了很多新的安全功能，使实施对被保护资源的访问控制更容易。[6](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml%22%20%5Cl%20%226)在 Java 早期的版本中，JVM 资源访问通过“沙箱”安全模型来实施。扩展通常限制到由平台提供程序（例如，浏览器、Web 服务器）所实现的功能。新的 JDK 1.2 权限模型更加灵活，甚至允许将应用程序定义的资源添加到访问控制系统中。现在，Java 程序能够定义对于敏感资源的访问限制，而不需要写新的 Security Manager 或修改底层平台。这意味着下载到浏览器中的 applet，或者装入到 Java 服务器中的 servlet 能够添加对 JVM 的资源访问控制，而不必修改底层浏览器或服务器实现。新安全模型的一个值得注意的特征是大多数访问控制实现包含在 Java 安全子系统中。通常，Java 程序（例如，applet、srvelet）和组件或库（例如，包、bean）不需要包含任何访问控制代码。当一个程序想要添加 JVM 的被保护资源时，可以添加一个方法调用，这个方法调用将检查这种受限操作是否得到允许。JDK 1.2 中采用的一种通用技术是创建一个被防护对象，藉此，访问对象或操作对象受到访问控制调用的限制。怎样使用访问控制功能的示例将在本篇论文的后面部分说明。JDK 1.2 访问控制子系统引入了一些新概念。第一个是 CodeSource，它是一组签发者（数字证书）和一个 codebase URL（统一资源定位符）的组合。CodeSource 是很多权限和访问控制决策的基础。第二个概念是安全策略。这个策略包含很多授权条目，这些条目描述授予特殊 CodeSource 的权限（请稍后参阅 Policy Database 旁注）。一个授权条目可以包含一个或多个 Permission，它是访问或使用被保护资源和被防护对象的权利。最后，ProtectionDomain 是 CodeSource 和在策略数据库中指定的，授予 CodeSource 的 Permission 的聚合体。通过 ClassLoader 装入到 JVM 的每个类文件被分配给 ProtectionDomain，由类的 CodeSource 确定。装入 Java 程序JVM 安全的三个支柱是 ByteCode Vertifier、Security Manager 和 ClassLoader。在 JDK 1.2 之前，每个应用程序必须写它自己的 Security Manager 和 ClassLoader 的子类。通过创建叫做 SecureClassLoader 的 ClassLoader 的子类，JDK 1.2 简化了开发过程。Security Manager 不再是抽象的，并且可以被实例化或生成子类。现在它的大多数方法调用类 AccessController 中的方法，该类在 JDK 1.2 中提供访问控制功能。因为大多数 Security Manager 方法调用 AccessController，这大大简化了新的 Security Manager 子类的编写。为了自动调用新的安全子系统，从本机操作系统的命令行启动 Java 应用程序。Java 运行时创建 SecureClassLoader 的一个实例，它接着用来定位和装入应用程序的类文件。SecurityManager 的子类在 Java 运行时被创建和安装。然后以命令行参数调用应用程序的 main() 方法。为启动 Java 应用程序，Java 运行时中的更改有两方面的目的。第一，在使用新 Java 安全访问控制子系统中安装简单的 Security Manager。第二，用 SecureClassLoader 安全并正确地将类装入到 Java 运行时中。SecureClassLoader 有几个重要的目的。第一是确保以正确的顺序完成对类的搜索。当 JVM 需要一个类时，SecureClassLoader 首先查看由 JVM 的类路径所引用的文件是否可用。类路径中的文件特指完全可信的类，是 Java 运行时的一部分。例如，随 JVM 一起提供的所有代码包含在类路径中，因此被认为是可信的代码。如果在类路径中没有，可以搜索应用程序定义的位置（例如，URL 请求中的 Web 服务器）。最后，代码可以是“Java 标准扩展”（Java Standard Extension）的一部分，是在主机文件系统中可用的一组类，但不是 JVM 类路径的一部分。“标准扩展”中的类通常位于主机系统（例如，工作站或个人计算机）的磁盘驱动器上，但这些类不是 JVM 的完全可信的运行时类的一部分。SecureClassLoader 第二个重要的目的是给装入到 JVM 中的类创建并设置 ProtectionDomain 信息。当 SecureClassLoader 将一个类装入到 JVM 中时，用来签发类文件的 codebase URL 和数字证书（如果存在）用来创建 CodeSource。CodeSource 用来定位（或实例化）类的 ProtectionDomain。ProtectionDomain 包含授予类的 Pemission。一旦类文件被装入到 JVM 中，SecureClassLoader 就将恰当的 ProtectionDomain 分配给类。这个 ProtectionDomain 信息以及特别是 ProtectionDomain 中的 Pemission 用来确定运行时期间的访问控制。一旦 Java 程序开始运行，SecureClassLoader 就帮助 JVM 装入运行程序所需要的其它类。这些类还根据它们的 CodeSource 分配到恰当的 ProtectionDomain。运行时访问控制在 Java 程序执行期间的不同点，请求访问被保护的资源。这样的访问包括（但不限于）网络 I/O 尝试、本地文件 I/O，或者尝试创建新的 ClassLoader 或访问一个程序定义的资源。为了验证是否允许正运行的程序执行操作，库例程调用 Security Manager 的 checkPermission（permissionToCheck）方法，该方法接下来调用 AccessController.checkPermission（permissionToCheck）。这些方法调用负责确定当前的线程是否具有足够的权限。checkPermission() 以 Permission 对象为参数。AccessController 方法 checkPermission() 回过头来检查当前线程的堆栈帧，获得线程的堆栈中的每个类的 ProtectionDomain（请参阅“线程堆栈帧”这一节）。当定位了堆栈帧中的每个 ProtectionDomain，permissionToCheck 与 ProtectionDomain 中包含的 Permission 进行比较。对于每个堆栈帧，如果 permissionToCheck 与 ProtectionDomain 的一个 Permission 匹配，那么继续用堆栈中下一个堆栈帧（类）的 ProtectionDomain 测试 Permission。这种测试一直重复到到达堆栈底为止。也就是说，线程中所有的类都有执行操作的权限。这样，访问控制检查成功通常意味着所请求的操作能够继续下去。如果没有授予 permissionToCheck 堆栈上所有类的权限（在类的所有 ProtectionDomain 中没有恰当的 Permission），那么抛出一个 SecurityException，并且拒绝访问资源。上面方案的一个缺点是，在类有一组 Permission，并且不关心它的调用者可能是谁的时候，例如，安装在桌面计算机上的 JavaBean 需要从本地磁盘驱动器上读文件。bean 的类的 ProtectionDomain 有读这些本地文件的 Permission。但是，从调用 bean 的 Web 服务器装入的程序有一个没有本地文件读权限的 ProtectionDomain。一般，如果 bean 由从 Web 服务器装入的程序调用，bean 将被拒绝访问本地磁盘驱动器上的文件，因为来自 Web 服务器的程序没有本地文件读 Permission。但是，如果 bean 调用 AccessController.beginPrivileged()，那么在线程的堆栈帧上作一个注释，表明当 AccessController.checkPermission（permissionToCheck）搜索 ProtectionDomain 时，搜索在这个堆栈帧处停止。bean 可能发出很多方法调用，但是当调用 AccessController.checkPermission（anotherPermissionToCheck）时，为找到 ProtectionDomain 往回遍历整个堆栈帧的搜索在这个堆栈帧处停止。基于上面的方案，将检查 bean 的 ProtectionDomain，但不会检查来自 Web 服务器的程序的 ProtectionDomain，因为搜索在 bean 的堆栈帧处停止。因此，文件读操作将会成功。为了关闭这种有特权的操作模式，对 AccessController.endPrivileged() 的调用除去了堆栈注释。如果应用程序忘记调用 AccessController.endPrivileged()，JVM 会得体地弥补，因为 beginPrivileged() 和 endPrivileged() 调用与堆栈帧相关联。也就是说，只要调用 beginPrivileged() 的方法存在，那么将自动关闭有特权的模式。上面的 beginPrivileged()/endPrivileged() 操作很巧妙的一面是，当新线程创建时，创建新线程的程序将丢失 ProtectionDomain 信息。也就是说，每个新线程创建一个新的运行时堆栈。父线程中的类在新线程中不存在。当 checkPermission() 操作完成时，重要的 ProtectionDomain 信息不再可用。与创建新线程的线程相比，这将给新线程更多的权限。为了避开这种安全信息的明显损失，当创建了子线程时，父线程的 ProtectionDomain 将附加在子线程上（由子线程继承）。这样，除非在子线程上完成 beginPrivileged() 操作，否则在 checkPermission() 操作期间还要检查父线程的 ProtectionDomain。线程堆栈帧JVM 中的每个线程包含很多堆栈“帧”。简单的说，这些帧包含当前线程中调用的每个方法的方法实例变量。如果使用了程序调试器，调试器将能够展示堆栈中每个方法的实例变量。为了进一步说明清楚，下面提供了两个示例。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image3.gif示例 1：当前线程的简单检查。图 1 展示了一个叫做 MyProgram 的程序的快照，其 codebase URL 是 http://www.NominallyWidgets.com。在初始化安全子系统之后，程序试图通过调用 System.getProperty("java.home") 来获取系统属性。getProperty 方法调用 Security Manager 方法 checkProperty()，来看看是否允许当前线程读系统属性。接着，Security Manager 用 PropertyPermission("java.home", "read") 为参数调用方法 AccessController.checkPermission，查看是否堆栈中所有的类都有恰当的权限。请注意本文中写到，没有将通过 JVM 类路径装入的类分配给 ProtectionDomain。这些类逻辑上被分配给系统域，系统域具有不受限制的对所有 Java 和应用程序定义的资源的访问权。也就是说，当 AccessController 进行检查时，它假设系统域类具有所有的权限。这个示例中的访问控制像下面这样工作：* 类 java.security.AccessController 在系统域中。缺省的情况下，系统域具有读属性的隐式权限；允许继续检查下一个堆栈帧。
* 类 java.lang.SecurityManager 在系统域中。缺省的情况下，系统域具有读属性的隐式权限；允许继续检查下一个堆栈帧。
* 类 java.lang.System 在系统域中。缺省的情况下，系统域具有读属性的隐式权限；允许继续检查下一个堆栈帧。
* MyProgram 有一个 codebase 为 http://www.NominallyWidgets.com 的 ProtectionDomain。要检查这个 ProtectionDomain 的权限。如果没有授予权限，将抛出一个安全异常，getProperty() 方法调用将失败。如果授予权限，允许检查下一个堆栈帧。在这个示例中，因为授予了权限，因此允许继续检查下一个堆栈帧。
* MyProgram 有一个 codebase 为 http://www.NominallyWidgets.com 的 ProtectionDomain。要检查这个 ProtectionDomain 的权限。如果授予这个权限，允许继续检查下一个堆栈帧。
* 类 java.lang.Thread 在系统域中。因为系统域具有读属性的隐式权限，所以允许继续检查下一个堆栈帧。

在这个示例中，操作是被许可的，因为成功检查了所有堆栈帧。如果这个线程由具有任何堆栈帧中的 ProtectionDomain 的另一个线程创建，那么还将检查从父线程继承的 ProtectionDomain。显然，优化还有一定空间，因为线程堆栈中的很多 ProtectionDomain 不是独一无二的。事实上，每次调用 checkPermission()，每个唯一的 ProtectionDomain 中的 Pemmission 只被检查一次。示例 2：调用 beginPrivileged()。在这个示例中，创建了一个线程，程序调用一个包含被保护资源的 bean（MyBean）。MyBean 调用 AccessController.beginPrivileged()，然后调用 MyBeanProtected 中的方法，这个方法将检查线程是否具有恰当的权限。线程的堆栈如图 2 所示。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image4.gif访问控制像下面这样工作：* java.lang.SecurityManager 是系统域的一部分，因此它有访问资源的隐式权限。接下来是下一个堆栈帧。
* MyBeanProtected 是具有资源访问权的 ProtectionDomain 的一部分。接下来是下一个堆栈帧。
* MyBean 是具有资源访问权的 ProtectionDomain 的一部分。因为从这个堆栈帧调用 AccessController.beginPrivileged()，因此在这里停止；不再检查堆栈帧。

下面的两阶段运算法则描述了 AccessController 计算权限是怎样的？第一阶段是获取一个第二阶段将使用的 ProtectionDomain 列表，如图 3 所示。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image5.gif第二阶段检查每个 ProtectionDomain，看它是否包含被检查的权限（图 4）。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image6.gif如果没有抛出异常，所请求的操作是被允许的。图 5 和图 6 中的流程图图示了刚才所描述的新功能怎样联系起来。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image7.gifhttp://www.personal.kent.edu/~hli4/tech/javasecurity.files/image8.gif工具JDK 1.2 中有三个重要的安全工具和两个数据资源库。这些工具主要是面向 JDK 用户，而不是面向包含 Java 的应用程序的最终用户。下面对它们进行了描述。Java 压缩文档（JAR）实用程序工具（这里表示为 jar）主要为了使将 Java 对象文件和资源打包到单个文件或压缩文档中更容易。当 Java 组件（类、图像文件、音频文件等）被放在一个压缩文档中时，它们可以通过与服务器的单个 HTTP（超文本传输协议）事务来下载，而不是为每个下载组件请求各自的 HTTP 连接。网络下载性能整体上得到了改进，特别是因为 jar 工具可以压缩压缩文档的内容。密钥管理工具 keytool 是一个密钥和证书管理实用程序。它与 jarsigner（请参阅下面的内容）配合，取代了 JDK 1.1 javakey 工具。keytool 实用程序使开发者能够管理他们自己的公钥和私钥对以及相关的证书，它们在需要数字签名的客户机认证或数据完整性和认证服务中使用。这个实用程序还允许高速缓存通信方（例如，Web 服务器）的公钥。Keytool 管理私钥的 keystore（资源库）和认证相应公钥的相关的 X.509 证书链。可以用 passphrase（在 JavaSoft 的缺省实现中）或通过更强大的保护机制（比如，密码术）来保护 keystore。keystore 中有两个基本的条目：* 密钥或证书条目 — 由私钥和证书链组成
* 可信的证书条目 — 带有公钥条目的多个单科证书

Keytool 可以创建 keystore，克隆或删除 keystore 中的条目，导入证书（可信的和不可信的），导出证书，显示 keystore 的内容以及生成自签发证书（包括公钥或私钥对）。keytool 的 Java 1.1 和 1.2 实现只支持 DSA（数字签名算法，Digital Signature Algorithm）密钥和 DSA/SHA-1（安全散列算法，Secure Hash Algorithm）签名算法。密钥大小的最大值限制为 1024 位。至于向下兼容性，人们期望 JDK 1.2 能够解析或处理 JDK 1.1 keystore 格式。数字签发 JAR 文件的实用程序 jarsigner 有两个主要的功能：签发 jar 文件以及验证所签发的 jar 文件的签名和完整性。jarsigner 用 jar 文件中所包含的证书来验证数字签名以及验证证书的公钥是否包含在所指定的 keystore 中。jarsigner 还验证 jar 文件没有以任何方式被篡改过。目前，jarsigner 只能签发用 jar 实用程序创建的 jar 文件。Jarsigner 可以使用与 SHA-1 摘要算法相结合的 DSA 密钥算法（缺省的加密引擎提供程序提供 DSA/SHA-1算法）或与 MD5（修改检测）摘要算法相结合的 RSA（源于最初的创立者：Rivest、Shamir 和 Adleman）密钥算法。也就是说，如果签发者的公钥和私钥是 DSA，jarsigner 使用 DSA/SHA-1 算法。如果提供了 RSA 密钥，则使用 RSA/MD5 算法。多次签发 jar 文件只需通过多次运行 jarsigner 就可以了，每次指定不同的签发者。访问控制策略数据库管理由 policytool 进行处理，policytool 是一种帮助用户（比如系统管理员）指定、生成、编辑、导出或导入 JVM 的安全访问控制策略的图形用户界面。这种工具创建 policyfile，在下面有描述。如前面所述，keystroe，密钥存储是私钥以及相关 X.509 证书链（认证相应的公钥）的资源库。keystore 是 java.security 包中所提供的 keystore 类的具体实现。上面描述的所有三个实用程序都使用 keystore。指定来自不同代码源的代码可用哪种权限的 Java 运行时策略由 Policy（访问控制策略数据库）对象表示。更确切地说，它由提供 Policy 类（它在 java.security 包中定义）中抽象方法的实现的 Policy 子类表示。Policy 的叫做 PolicyFile 的缺省实现从平面 ASCII 文件读策略信息。策略框架允许将策略信息存储在本地系统或网络中的任何地方。Java 1.2 安装的策略配置文件指定来自不同代码源的代码允许什么权限（哪种类型的系统资源访问权）。存储在策略数据库的信息在“Policy Database”旁注中有描述。http://www.personal.kent.edu/~hli4/tech/javasecurity.files/image9.gifJava 和基于浏览器的安全模型浏览器和其它因特网技术比 Java 的广泛引入早进入市场；因此，由快速演进的 Java 的相关技术提供的安全模型中存在失配不足为奇。如果模型选择的部分可以按有序的方式达成协议，那么朝着同步方向发展可能取得巨大进步。这种失配主要源于早期 Java 安全的局限性以及填补这些空白来满足客户的需求的愿望。模型调整中协调工作的这样一个方面是在访问控制模型的协调。这一节提供了具有代表性的浏览器的功能类的高级观点，这些浏览器具有 Java 1.2 安全体系结构中的可比较的功能。都使用基于堆栈的方法来进行授权。但是，它们使用了截然不同的访问控制模型和授权机制。浏览器的提案更是雄心勃勃，但是所添加的功能可能更难以管理。应该认识到失配在浏览器安全模型中也存在，因为它们将支持扩展为包括对“超文本标记语言”（HTML）页面、脚本和 applet 的容纳。在一些环境中，被签发的 Java applet 将减少对一些 Java 元素的访问，因为调用 applet 的包含页面或脚本没有被签发。下面讨论将只突出浏览器的 Java 访问控制机制。另一种访问控制模型商业上可用的 Web 浏览器的“Java 虚拟机”实现被仔细研究过了，该实现使用基于目标和权限的显式激活或取消激活的另一种访问控制模型。目标是将主体映射到对象上的操作（也就是，大致是传统的权限表示）。能否启用权限取决于“启用”策略的三值逻辑。本质上，如果至少一个主体（用户、系统管理员、目标类定义者）允许这个目标，并且没有主体禁止这个目标，那么可以启用这个目标。Java 开发者可以在运行时启用这样的目标。开发者还可以在运行时禁用（禁止）或还原（撤消启用）目标。这个模型启用了合并多个主体和授予 applet 低于最高权限的权限的策略。由 applet 代码来管理所启用权限的子集。在运行时，目标由 enablePrivilege() 激活，由 disablePrivilege() 禁止以及由 revertPrivilege() 取消激活。还原只影响调用的堆栈帧，因此前面堆栈帧启用的目标仍然激活，并取代还原。因此，子代方法除去特权是不可能的，否则可能造成安全漏洞。对于请求，浏览器授权机制检查启用的特权的堆栈。如果找到一个堆栈，不管线程的堆栈中的类（方法的调用者）的信任是什么样的，操作都会被允许。因此，权限的显式禁止或恰当的权限还原对于阻止未授权主体使用另一个主体“启用的”特权来说是必需的。请注意，在线程堆栈中可能有启用更高权限的其它方法。如果 Java 核心类（基本的 Java 运行时）中的方法需要添加访问控制来关闭安全漏洞，那么这一节中所描述的浏览器模型将给现有的代码带来麻烦。添加一个权限从而允许代码启用这个权限很简单。但是，代码已经不包含启用这个权限的代码，因为当最初写编码时不需要权限。同样的情况对于库和 bean 编写者同样成立，这些编写者在他们的软件的后续版本中发现他们需要添加访问控制。其含义是应用程序编写者将不得不更新他们的代码，以支持（也就是启用）任何添加到他们使用的基本 Java 类或库和 bean 的新权限。这导致了严峻的版本管理问题。JDK 1.2 访问控制模型不存在这个问题。在 JDK 1.2 中，只需要对策略数据库进行更新。总之，浏览器模型比 Java 模型提供了更多的功能，而这个功能赋予程序员跟踪授予的和撤消的权限，了解浏览器、JVM 或库／bean 版本差异的更多的责任。基于堆栈的授权Java 当前的访问控制机制基于“栈内省”，也就是逻辑地遍历线程的堆栈帧，看调用的方法或类是否有足够的权限来完成所请求的操作。鉴于当前权限结构的粗糙粒度，基于堆栈的授权的性能看起来是可接受的。但是，因为所有的安全问题不能都变为栈内省，所以一个对象将权限传递给另一个对象，并获得用别的方法不能直接获得的信息是可能的。例如，有可能促使另一个对象将权限（比如，文件描述符）传递给未授权的对象。当创建文件描述符时，未授权的对象不在堆栈中，而它仍然接收了对象引用（“离栈”欺骗，“off-the-stack”snoofing）。这种类型的安全攻击是基于堆栈的授权技术所特有的，因为这种技术不跟踪对象之间不安全的交互。在那些具有严格通信要求（比如，要求分等级、网格状、关于信息流的通信协议 ）的环境中，Java 安全模型可能还不够。但是，对于大多数能预见的 Java 用途而言，基于栈内省的访问控制功能对于实现任务关键的应用程序已经足够了。安全要求 — 高级观点下面的要求将满足被认为超出 JDK 1.2 中目前计划提供的功能级别的需要；就其中的很多要求，正在与 JavaSoft 进行讨论，一些要求可能在 JDK 1.2 发行时或后续的 JDK 发行版中满足。这些是具有代表性的类别以及每个类别中的要求示例，不必是每个类别中已知要求的完整汇集。* **Java 虚拟机高完整性计算环境：**这个要求支持具有多组安全凭证的并发的 applet 或 servlet 执行。因为系统之间（有时是子系统之间）认证和凭证要求不同，JVM 内的 applet 或 servlet 支持处理多组安全凭证是有必要的。简化的 API（应用程序编程接口）将使应用程序编写者发掘这些安全功能更容易。

满足这个要求使电子交易应用程序所需要的客户机和服务器以及服务器子系统之间能够进行交互。* **策略驱动的 Java 安全模型和安全服务：**客户应该能够定义和部署安全策略。底层的系统（比如，Java）应该具备足够的机制来实现和实施策略。实施策略的机制需要包括对以下这些方面的支持：访问控制、加密和保护级别、信任、安全将策略语句传送给 JVM、策略管理以及策略引擎的 JVM 支持。

满足这个要求可以通过更简单的配置和策略管理减少所有权的总成本，更简单的配置和策略管理通过使 JVM 成为所有 Java 应用程序的策略实施的单点来实现。* **简单的安全编程模型：**需要针对保护级别（保密性、完整性和不可抵赖性）的简单、高级的 API，使没有安全意识的应用程序获得缺省的安全保护，比如像安全通信、安全文档或邮件、安全流以及安全的远程方法调用（RMI）和因特网 ORB 间协议（IIOP）这样的功能。[7、8](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml#9)

这个要求可以简化编程模型，使安全权限对于任何应用程序都容易使用，并减少对安全无知的程序员所造成的潜在错误。* **应用程序安全部署的标准：**针对 applet 和应用程序交付的清单格式和单一签名标准（比如 W3C）是需要的。

这个要求应该建立确保对服务器和客户机系统的应用程序交付的单一和低成本的方式。* **标准化编程模型：**创建建立由多个嵌入在 HTML 或 XML（可扩展标记语言）文档中的 applet 组成的应用程序的信任和完整性的标准。

这个要求可以给应用程序提供分配访问控制或其它策略的单一模型，应用程序可能由多个组件化元素组成。* **本机安全服务支持：**利用底层平台的安全功能。

这个要求将最大化平台安全功能，改善性能并考虑到提供一致性。* **（加密的）服务提供程序开发和部署的标准：**加密的服务提供程序应该被签发，并在提供程序中提供服务的枚举描述。

这个要求应该确保满足国际化的部署要求，并且需要这些服务的应用程序可以访问所需的信息。控制加密操作的强度（比如，密钥长度、签名长度、算法长度等）可以将加密服务进行国际化部署。* **可维护性、可伸缩性和互操作性：**提供集中管理和应变能力、交互操作和利用非 Java 安全功能的能力以及客户机和服务器之间所需要的以分布式方式支持所有安全功能的能力。

这个要求应该考虑到受控制的 Java 部署，并提高性能和迁移性。* **使安全免于成为性能的阻碍：**允许在类文件的验证期间使用算法的硬件支持或本机支持实现，并允许策略定义：在类装入时，可信签发者的组合和可信编译器的使用将允许覆盖动态字节码验证。

很多安全功能是极度性能密集的（比如，散列、密钥生成）；需要进行改进，以达到非 Java 环境中的性能。应该注意 Java 不独立运行；Java 在操作系统平台的上下文中运行，它在这种平台上被实现。此外，Java 经常被嵌入到另一个应用程序中，比如 Web 浏览器或 Web 服务器。这些操作系统和子系统的每一个都影响着 JVM 和 Java 运行时的安全攻击的弱点。[8 - 11](http://www-900.ibm.com/developerWorks/cn/java/j-javaevol/index.shtml#10)当部署 Java 程序时，应该注意配置 JVM 和应用程序文件来最小化安全攻击的弱点；这方面的讨论超出了本文的范围。未来的方向由于软件业持续的强大支持，使 Java 环境符合今天非 Java 环境通常提出的最严格的要求所需要的很多改进措施成为可能。这些功能包括强大的加密、成熟的访问控制以及提供集中安全策略管理的能力。这其中的一些功能很可能在 1998 年中期至 1999 全年可开始使用。为提供访问现有工业级的安全机制的 Java 扩展，还需要进行很多工作，从而提高系统完整性、性能以及功能。将来，我们还将看到高级电子交易和其它类型的应用程序，从单独提供的安全功能转变为利用 JDK 最新版本中的功能。到 1998 年末或 1999 年初，有望在所有的 Java 环境中实现具有重要意义的最初的安全功能。IBM、Lotus 和 Tivoli 正在积极和 Java 软件业一起提出业务关键的企业安全要求，并确保它们被满足。IBM 的 Java 倡议敏锐地意识到需要把 Java 环境的安全作为重点。  |  |