1. Java安全
2. 概述

Java技术提供了三种确保安全的机制：

1. 语言设计特性（数组边界检查，类型转换检查）
2. 访问控制机制，用于控制代码能够执行的操作（文件访问，网络访问）
3. 代码签名，代码的作者能够用标准的加密算法来认证Java代码，检查代码是否被修改过
4. 类加载器

Java编译器会为虚拟机转换源指令，虚拟机代码存储在.class为扩展名的类文件中，每个类文件都包含某个类或者接口的定义和实现代码。

1. 类加载过程

虚拟机只加载程序执行时所需要的类文件，首先读取类文件的内容，如果该类拥有另一个类的域或是超类，那么这些类文件也会被加载，加载某个类所依赖的所有类的过程称为类的解析。接着，虚拟机加载类中的main方法，如果main方法要用到更多的类，那么就会加载这些类。

1. 类加载器的层次结构

每个Java程序至少有三个类加载器：引导类加载器，平台类加载器，系统类加载器。

引导类加载器负责加载JDK内部模块中的平台类，该加载器没有对应的ClassLoader对象。Java9之前，Java平台类位于rt.jar中，如今的Java平台是模块化的，每个平台模块都包含一个JMOD文件，平台类加载器会加载引导类加载器没有加载的Java平台中的所有类。

系统类加载器会从模块路径和类路径中加载应用类。

类加载器有父子关系，除了引导类加载器外，每个类加载器都有一个父类加载器，每个父类加载器会首先加载，当父类加载器加载失败时，它才会加载该给定的类。某些插件类也是可以被加载的，使用URLClassLoader类的实例来加载插件类，由于在URLClassLoader构造器中没有指定父类加载器，因此pluginLoader的父亲就是系统类加载器。

类的加载是由于其他的类需要它而被加载的，偶尔，我们也需要干涉和指定类加载器，如加载一个助手方法：

Class.forName(classNameString);

这个方法是从一个插件类中调用的，classNameString指定的是一个包含在这个插件中的JAR中的类，这个类通过Class.forName加载，使用的是系统类加载器，而对于它来说，插件JAR中的类是不可见的，这种现象称为类加载器倒置。因此，插件类需要使用恰当的类加载器，将指定的类加载器设置为当前线程的上下文类加载器，每个线程都有一个对类加载器的引用，称为上下文类加载器，主线程的上下文类加载器是系统类加载器，当新线程创建时，它的上下文类加载器会被设置为父线程的上下文类加载器。如下所示：

Thread t= Thread.currentThread();

t.setContextClassLoader(loader);

然后插件类可以获取这个上下文类加载器：

Thread t = Thread.currentThread();

ClassLoader loader = t.getContextClassLoader();

Class<?> cl = loader.loadClass(claaName);

1. 类加载器作为命名空间

在同一个虚拟机中，可以有两个类，他们的类名和包名可以是相同的，类是由它的全名和类加载器来确定的，例如，应用服务器会为每一个应用使用单独的类加载器，这使得虚拟机可以区分来自不同应用的类。

1. 字节码校验

当类加载器将新加载的Java平台类的字节码传递给虚拟机时，这些字节码首先要接受校验器的检验，校验器负责检查哪些指令无法执行的明显有破坏性的操作，除了系统类外，所有的类都要被检验。如：

1. 变量的初始化；
2. 方法调用与对象引用类型之间的匹配；
3. 对本地变量的访问都落在运行时堆栈内；
4. 运行时堆栈没有溢出；
5. 安全管理器

一旦类被加载到虚拟机中，并且有检验器检查过之后，Java平台的第二种安全机制就会启动，这个机制就是安全管理器。

1. 权限检查

安全管理器是一个负责控制具体操作是否允许执行的类，如退出虚拟机等等，在运行Java应用程序时，默认的设置是不安装安全管理器的，所有的操作都是默许的。我们可以设置一个策略，在遇到安全问题时通过策略中的对策来执行。但是，安全策略的完整性依赖于谨慎的编码，标准类库中系统服务的提供者，在试图继续任何敏感的操作之前，都必须与安全管理器进行协商。

1. Java平台安全性

从Java1.2开始，Java平台拥有了更加灵活的安全机制，它的安全策略建立了代码来源和访问权限集之间的映射关系。代码来源是由一个代码位置和一个证书集指定的，代码位置指定了代码的来源，如位于JAR包中的代码的代码位置是该文件的URL。证书的目的是要由某一方来保证代码没有被篡改过。权限是指由安全管理器负责检查的任何属性，Java平台支持许多访问权限类，每个类都封装了特定权限的详细信息。每个类都有一个保护域，它是一个用于封装类的代码来源和权限集合的对象，当SecurityManager类需要检查某个权限时，它要查看当前位于调用堆栈上的所有方法的类，然后它要获得所有类的保护域，并且询问每个保护域，其权限集合是否允许执行当前正在被检查的操作。

1. 安全策略文件

策略管理器要读取相应的策略文件，这些文件包含了将代码来源映射为权限的指令，下面是一个策略文件：

grant codebase <http://www.horstmann.com/classes>

{

permission java.io.FilePermission “/tmp/\*”, “read, write”;

}

1. 数字签名
2. 消息摘要

消息摘要是数据块的数字指纹，例如SHA1（安全散列算法#1）可以将任何数据块，无论其数据有多长，都压缩为160位的序列。如果数据的1位或者几位改变了，那么消息摘要也会改变。且拥有给定消息的伪造者无法创建与原消息具有相同摘要的假消息。

MessageDigest类是用于创建封装了指纹算法的对象的工厂，它的静态方法getInstance返回继承了MessageDigest类的某个类的对象，获取计算SHA指纹的对象如下所示：

MessageDigest alg = MessageDigest.getInstance(“SHA-1”);

获取到MessageDigest对象之后，可以通过反复调用update方法，将信息中的所有字节都提供给该对象。

1. 消息签名

消息摘要解决了消息的鉴定问题，当消息和消息摘要是分开传送的，接受者就可以检查消息是否被篡改过，但是如果消息和消息摘要同时被截获了，然后对消息进行修改，根据公开的消息摘要算法计算指纹，那么接受者永远不会知道消息已经被篡改了。消息签名解决了这个问题。

公共密钥加密技术是基于公共密钥和私有密钥这两个概念的，公钥可以公布给世界，但是私钥只能自己持有。几乎不可能通过一个密钥来推算出另一个密钥，也就是说即使每个人都知道你的公钥，但是不可能计算出你的私钥。因此，当A想要发送消息给B时，A用自己的私钥对消息摘要签名， 而B知道A的公钥，B用A的公钥对该签名进行校验，如果通过了校验，则B知道原始消息没有被篡改，且该消息是由A签名的，因为它是私钥的持有者。但是这也隐含了风险，如果A的私钥被人偷走了，那么A的身份就会其他人替代。

1. 校验签名

JDK配有一个keytool程序，该程序是一个命令行工具，用于生成和管理一组证书，以及管理密钥库，证书数据库和私有/公有密钥对，密钥库中的每一项都有一个别名，下面是Alice创建一个密钥库alice.certs并且用别名生成一个密钥对：

keytool -genkeypair -keystore alice.certs -alias alice

创建完密钥库后，Alice可以导出公钥：

keytool -exportcert -keystore alice.certs -alias alice -file alice.cer

当接收者收到公钥，并导入密钥库：

keytool -importcert -keystore bob.certs -alias alice -file alice.cer

Alice就可以给接受则发送签过名的文档了，jarsigner工具负责对JAR文件进行签名和校验，Alice只需要将文档添加到要签名的JAR文件中即可：

jar cvf document.jar document.txt

然后使用jarsigner工具将签名添加到文件中，指定要使用的密钥库，JAR文件和密钥的别名：

jarsigner -keystore alice.certs document.jar alice

当接收者收到JAR文件后，使用jarsigner的-verify选项对文件进行校验：

jarsigner -verify -keystore bob.certs document.jar

1. 认证问题

我们之前解决的都是消息的校验问题，即消息是否还是原来的消息我们可以确定，但是还有发送消息者的身份问题的认证没有解决。只有我们信任发送消息者的身份，我们才能够认证他的消息。其实就是一个信任问题，如果我们只信任A发送的消息，只要陌生人将公钥发送给A，A在公钥的基础上用私钥签名。我们持有A的公钥，只需要用A的公钥对A发送来的消息校验，就可以确认A已经核实了陌生人的身份。

但是，通常情况下不会有这样一个熟人身份，一般都是一个信任链条，我们可以信任链条中的任何一个人；或者是一个大家都信任的机构，提供认证服务。我们常常会遇到由负责担保他人身份的一个或多个实体签署的数字签名，但是我们需要评估在多大程度上能够信任这些机构。

５.　证书签名

互联网上的信息传递是非常频繁的，我们不可能对每一条信息都进行校验，那么过于繁琐，因此如果有一个我们信任的实体来校验这些签名是最好的。比如说Cindy信任ACME软件部，这个部门负责证书授权（CA）的运作，首先我们创建一个密钥库acme.Certs，生成一个密钥对并导出公共密钥，其中的公共密钥被导入到一个自签名的证书中：

keytool -genkeypair -keystore acme.certs -alias acmeroot

keytool -exportcert -keystore acme.certs -alias acmeroot -file acmeroot.cer

如果Alice要发送消息给Cindy，他需要将自己的证书签名，并提交给ACME，ACME将负责审核Alice的身份，并且生成了签名证书，Alice将该证书交给Cindy或者放在ACME公司的目录上，该文件包含了Alice的公钥和ACME的声明，证明该密钥确实属于Alice。这样，Cindy将该证书导入到她的密钥库中后，密钥库校验该密钥是由密钥库中已有的受信任的根密钥签名过的，Cindy就不需要对证书的指纹进行校验了。一旦Cindy添加了根证书和经常给他发送消息的人的证书后，就不用担心密钥库了。

1. 加密
2. 概述

当信息通过认证后，信息本身是可见的，加密技术能够对信息进行加密，只能用匹配的密钥进行解密。认证对于代码签名是足够的， 没有必要将代码隐藏起来，而对于机密消息，需要进行加密。

1. 对称密码

Java密码扩展包含了一个Cipher类，该类是所有加密算法的超类，调用getInstance方法获得一个密码对象：

Ciphre cipher = Cipher.getInstance(aligorithmName);

算法通常有“DES”，或者“AES”等等，DES是数据加密标准，AES是高级加密标准，一旦获得了一个密码对象，就可以通过设置模式和密钥来对它初始化。

1. 密钥生成

为了加密，我们需要生成密钥，每个密码都有不同的用于密钥的格式，我们需要确保密钥的生成是随机的，需要遵循以下的步骤：

1. 为加密算法获取KeyGenerator；
2. 用随机源来初始化密钥发生器；
3. 调用generateKey方法；

如生成AES密钥：

KeyGenerator keygen = KeyGenerator,getInstance(“AES”);

var random = new SecureRandom();

keygen.init(random);

Key key = keygen.generateKey();

如果要生成密钥，必须用真正的随机数，Random类中常规的随机数发生器，是根据日期和时间来产生随机数的，不够随机，因为计算机时钟可以精确到0.1秒，每天最多存储864000个种子，如果攻击者知道发布密钥的日期，就可以生成那一天所有可能的种子。

SecureRandom类产生的随机数更加安全，但是仍然要提供一个种子，以便在一个随机点上开始生成数字序列。可以从硬件设备或者随机敲击键盘获得。

所有已知的公共密钥算法的速度都比对称密钥算法（AES或DES）慢得多，使用公共密钥算法对大量信息加密是不合适的，可以用公共密钥算法加密少量关键数据，用对称密钥算法加密大量一般数据。