**您的 Java 代码安全吗 — 还是暴露在外？**

**现在构建更安全的应用程序以避免以后的麻烦**

虽然客户仍然很关心您为他们构建的应用程序的可伸缩性和可用性，但他们可能变得也很关心安全性，而且要求特别严格。应用程序可能容易受到两类安全性威胁的攻击：静态和动态。虽然开发人员不能完全控制动态威胁，但在开发应用程序时，您可以采取一些预防措施来消除静态威胁。本文概括并解释了 13 种类型的静态暴露 ― 它们是系统中的缺陷，它使系统暴露在想要篡夺该系统的特权的攻击者面前。您将学会如何处理这些暴露，以及如何发现（如果不处理这些暴露）这些暴露可能造成的影响。

1http://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#icomments)

[Bijaya Nanda Sahu](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#authorN1001A), 软件工程师, IBM Global Services，印度

[Srinivasan S. Muthuswamy](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#authorN1002B)([smuthusw@in.ibm.com](mailto:smuthusw@in.ibm.com?subject=%E6%82%A8%E7%9A%84%20Java%20%E4%BB%A3%E7%A0%81%E5%AE%89%E5%85%A8%E5%90%97%20%E2%80%94%20%E8%BF%98%E6%98%AF%E6%9A%B4%E9%9C%B2%E5%9C%A8%E5%A4%96%EF%BC%9F&cc=smuthusw@in.ibm.com)), 软件工程师, IBM Global Services，印度

[Satya Nanaji Rao Mallampalli](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#authorN1003C)([msatyana@in.ibm.com](mailto:msatyana@in.ibm.com?subject=%E6%82%A8%E7%9A%84%20Java%20%E4%BB%A3%E7%A0%81%E5%AE%89%E5%85%A8%E5%90%97%20%E2%80%94%20%E8%BF%98%E6%98%AF%E6%9A%B4%E9%9C%B2%E5%9C%A8%E5%A4%96%EF%BC%9F&cc=msatyana@in.ibm.com)), 软件工程师, IBM Global Services，印度

[Venkata R. Bonam](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#authorN1004D) ([vbonam@in.ibm.com](mailto:vbonam@in.ibm.com?subject=%E6%82%A8%E7%9A%84%20Java%20%E4%BB%A3%E7%A0%81%E5%AE%89%E5%85%A8%E5%90%97%20%E2%80%94%20%E8%BF%98%E6%98%AF%E6%9A%B4%E9%9C%B2%E5%9C%A8%E5%A4%96%EF%BC%9F&cc=vbonam@in.ibm.com)), 资深软件工程师, IBM Global Services，印度

2002 年 9 月 01 日

* [expand](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-j-staticsec-_-sidebar)

在开发 Java Web 应用程序时，您需要确保应用程序拥有完善的安全性特征补充。这里在谈到 Java 安全性时，我们并不谈及 Java 语言提供的安全性 API，也不涉及使用 Java 代码来保护应用程序。本文将着重讨论可能潜伏在您的 Java 应用程序中的 *安全性暴露*。安全性暴露是系统中的缺陷，它使系统无法 ― 即使系统被正常使用 ― 防止攻击者篡夺对系统的特权、控制系统的运行、危及系统上的数据安全或者假冒未经授权的信任。相对于安全性暴露，许多开发人员更加关心网站的感官效果。

毫无疑问，客户现在既严格地关注性能、可伸缩性和可用性也严格地关注安全性。应用程序可能容易受到两类安全性威胁的攻击： *动态*和 *静态*。动态威胁是那些同未经授权进入系统有关的威胁，或那些同跨越网络传输的数据的完整性、隐私和机密性有关的威胁。这些威胁同应用程序的功能代码没有多大关系；使用加密、加密术和认证技术来消除这些威胁。相比之下，静态威胁却同应用程序的功能代码 *有关*；它们同进入系统的授权用户所做的事情有关。未知用户闯入系统是动态威胁的一个示例；授权用户以未授权方式操作系统内的代码或数据是静态威胁的示例。应用程序开发人员并不能完全控制动态威胁；但开发人员在构建应用程序时却可以采取预防措施来消除静态威胁。

在本文中，我们讨论了对付 13 种不同静态暴露的技巧。对于每种暴露，我们解释了不处理这些安全性问题所造成的影响。我们还为您推荐了一些准则，要开发不受这些静态安全性暴露威胁的、健壮且安全的 Java 应用程序，您应该遵循这些准则。一有合适的时机，我们就提供代码样本（既有暴露的代码也有无暴露的代码）。

**对付高严重性暴露的技巧**

请遵循下列建议以避免高严重性静态安全性暴露：

* [限制对变量的访问](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#2)
* [让每个类和方法都成为 final，除非有足够的理由不这样做](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#5)
* [不要依赖包作用域](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#8)
* [使类不可克隆](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#11)
* [使类不可序列化](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#14)
* [使类不可逆序列化](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#17)
* [避免硬编码敏感数据](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#20)
* [查找恶意代码](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#23)

**限制对变量的访问**

如果将变量声明为 public，那么外部代码就可以操作该变量。这可能会导致安全性暴露。

**影响**  
如果实例变量为 public，那么就可以在类实例上直接访问和操作该实例变量。将实例变量声明为 protected并不一定能解决这一问题：虽然不可能直接在类实例基础上访问这样的变量，但仍然可以从派生类访问这个变量。

清单 1 演示了带有 public 变量的代码，因为变量为 public的，所以它暴露了。

**清单 1. 带有 public 变量的代码**

class Test {

public int id;

protected String name;

Test(){

id = 1;

name = "hello world";

}

//code

}

public class MyClass extends Test{

public void methodIllegalSet(String name){

this.name = name; // this should not be allowed

}

public static void main(String[] args){

Test obj = new Test();

obj.id = 123; // this should not be allowed

MyClass mc = new MyClass();

mc.methodIllegalSet("Illegal Set Value");

}

}

**建议**  
一般来说，应该使用取值方法而不是 public 变量。按照具体问题具体对待的原则，在确定哪些变量特别重要因而应该声明为 private 时，请将编码的方便程度及成本同安全性需要加以比较。清单 2 演示了以下列方式来使之安全的代码：

**清单 2. 不带有 public 变量的代码**

class Test {

private int id;

private String name;

Test(){

id = 1;

name = "hello world";

}

public void setId(int id){

this.id = id;

}

public void setName(String name){

this.name = name;

}

public int getId(){

return id;

}

public String getName(){

return name;

}

}

**让每个类和方法都为 final**

不允许扩展的类和方法应该声明为 final。这样做防止了系统外的代码扩展类并修改类的行为。

**影响**  
仅仅将类声明为非 public并不能防止攻击者扩展类，因为仍然可以从它自己的包内访问该类。

**建议**  
让每个类和方法都成为 final，除非有足够的理由不这样做。按此建议，我们要求您放弃可扩展性，虽然它是使用诸如 Java 语言之类的面向对象语言的主要优点之一。在试图提供安全性时，可扩展性却成了您的敌人；可扩展性只会为攻击者提供更多给您带来麻烦的方法。

**不要依赖包作用域**

没有显式地标注为 public、 private或 protected的类、方法和变量在它们自己的包内是可访问的。

**影响**  
如果 Java 包不是封闭的，那么攻击者就可以向包内引入新类并使用该新类来访问您想保护的内容。诸如 java.lang之类的一些包缺省是封闭的，一些 JVM 也让您封闭自己的包。然而，您最好假定包是不封闭的。

**建议**  
从软件工程观点来看，包作用域具有重要意义，因为它可以阻止对您想隐藏的内容进行偶然的、无意中的访问。但不要依靠它来获取安全性。应该将类、方法和变量显式标注为 public、 private或 protected中适合您特定需求的那种。

**使类不可克隆**

克隆允许绕过构造器而轻易地复制类实例。

**影响**  
即使您没有有意使类可克隆，外部源仍然可以定义您的类的子类，并使该子类实现 java.lang.Cloneable。这就让攻击者创建了您的类的新实例。拷贝现有对象的内存映象生成了新的实例；虽然这样做有时候是生成新对象的可接受方法，但是大多数时候是不可接受的。清单 3 说明了因为可克隆而暴露的代码：

**清单 3. 可克隆代码**

class MyClass{

private int id;

private String name;

public MyClass(){

id=1;

name="HaryPorter";

}

public MyClass(int id,String name){

this.id=id;

this.name=name;

}

public void display(){

System.out.println("Id ="+id+"\n"+"Name="+name);

}

}

// hackers code to clone the user class

public class Hacker extends MyClass implements Cloneable {

public static void main(String[] args){

Hacker hack=new Hacker();

try{

MyClass o=(MyClass)hack.clone();

o.display();

}

catch(CloneNotSupportedException e){

e.printStackTrace();

}

}

}

**建议**  
要防止类被克隆，可以将清单 4 中所示的方法添加到您的类中：

**清单 4. 使您的代码不可克隆**

public final Object clone()

throws java.lang.CloneNotSupportedException{

throw new java.lang.CloneNotSupportedException();

}

如果想让您的类可克隆并且您已经考虑了这一选择的后果，那么您仍然可以保护您的类。要做到这一点，请在您的类中定义一个为 final的克隆方法，并让它依赖于您的一个超类中的一个非 final克隆方法，如清单 5 中所示：

**清单 5. 以安全的方式使您的代码可克隆**

public final Object clone()

throws java.lang.CloneNotSupportedException {

super.clone();

}

类中出现 clone()方法防止攻击者重新定义您的 clone 方法。

**使类不可序列化**

序列化允许将类实例中的数据保存在外部文件中。闯入代码可以克隆或复制实例，然后对它进行序列化。

**影响**  
序列化是令人担忧的，因为它允许外部源获取对您的对象的内部状态的控制。这一外部源可以将您的对象之一序列化成攻击者随后可以读取的字节数组，这使得攻击者可以完全审查您的对象的内部状态，包括您标记为 private的任何字段。它也允许攻击者访问您引用的任何对象的内部状态。

**建议**  
要防止类中的对象被序列化，请在类中定义清单 6 中的 writeObject()方法：

**清单 6. 防止对象序列化**

private final void writeObject(ObjectOutputStream out)

throws java.io.NotSerializableException {

throw new java.io.NotSerializableException("This object cannot

be serialized");

}

通过将 writeObject()方法声明为 final，防止了攻击者覆盖该方法。

**使类不可逆序列化**

通过使用逆序列化，攻击者可以用外部数据或字节流来实例化类。

**影响**  
不管类是否可以序列化，都可以对它进行逆序列化。外部源可以创建逆序列化成类实例的字节序列。这种可能为您带来了大量风险，因为您不能控制逆序列化对象的状态。请将逆序列化作为您的对象的另一种公共构造器 ― 一种您无法控制的构造器。

**建议**  
要防止对对象的逆序列化，应该在您的类中定义清单 7 中的 readObject()方法：

**清单 7. 防止对象逆序列化**

private final void readObject(ObjectInputStream in)

throws java.io.NotSerializableException {

throw new java.io.NotSerializableException("This object cannot

be deserialized");

}

通过将该方法声明为 final，防止了攻击者覆盖该方法。

**避免硬编码敏感数据**

您可能会尝试将诸如加密密钥之类的秘密存放在您的应用程序或库的代码。对于你们开发人员来说，这样做通常会把事情变得更简单。

**影响**  
任何运行您的代码的人都可以完全访问以这种方法存储的秘密。没有什么东西可以防止心怀叵测的程序员或虚拟机窥探您的代码并了解其秘密。

**建议**  
可以以一种只可被您解密的方式将秘密存储在您代码中。在这种情形下，秘密只在于您的代码所使用的算法。这样做没有多大坏处，但不要洋洋得意，认为这样做提供了牢固的保护。您可以 *遮掩*您的源代码或字节码 ― 也就是，以一种为了解密必须知道加密格式的方法对源代码或字节码进行加密 ― 但攻击者极有可能能够推断出加密格式，对遮掩的代码进行逆向工程从而揭露其秘密。

这一问题的一种可能解决方案是：将敏感数据保存在属性文件中，无论什么时候需要这些数据，都可以从该文件读取。如果数据极其敏感，那么在访问属性文件时，您的应用程序应该使用一些加密／解密技术。

**查找恶意代码**

从事某个项目的某个心怀叵测的开发人员可能故意引入易受攻击的代码，打算日后利用它。这样的代码在初始化时可能会启动一个后台进程，该进程可以为闯入者开后门。它也可以更改一些敏感数据。

这样的恶意代码有三类：

* 类中的 main 方法
* 定义过且未使用的方法
* 注释中的死代码

**影响**  
入口点程序可能很危险而且有恶意。通常，Java 开发人员往往在其类中编写 main()方法，这有助于测试单个类的功能。当类从测试转移到生产环境时，带有 main()方法的类就成为了对应用程序的潜在威胁，因为闯入者将它们用作入口点。

请检查代码中是否有未使用的方法出现。这些方法在测试期间将会通过所有的安全检查，因为在代码中不调用它们 ― 但它们可能含有硬编码在它们内部的敏感数据（虽然是测试数据）。引入一小段代码的攻击者随后可能调用这样的方法。

避免最终应用程序中的死代码（注释内的代码）。如果闯入者去掉了对这样的代码的注释，那么代码可能会影响系统的功能性。

可以在清单 8 中看到所有三种类型的恶意代码的示例：

**清单 8. 潜在恶意的 Java 代码**

public void unusedMethod(){

// code written to harm the system

}

public void usedMethod(){

//unusedMethod(); //code in comment put with bad intentions,

//might affect the system if uncommented

// int x = 100;

// x=x+10; //Code in comment, might affect the

//functionality of the system if uncommented

}

**建议**  
应该将（除启动应用程序的 main() 方法之外的） main()方法、未使用的方法以及死代码从应用程序代码中除去。在软件交付使用之前，主要开发人员应该对敏感应用程序进行一次全面的代码评审。应该使用“Stub”或“dummy”类代替 main()方法以测试应用程序的功能。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#ibm-pcon)

**对付中等严重性暴露的技巧**

请遵循下列建议以避免中等严重性静态安全性暴露：

* [不要依赖初始化](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#26)
* [不要通过名称来比较类](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#29)
* [不要使用内部类](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#32)

**不要依赖初始化**

您可以不运行构造器而分配对象。这些对象使用起来不安全，因为它们不是通过构造器初始化的。

**影响**  
在初始化时验证对象确保了数据的完整性。

例如，请想象为客户创建新帐户的 Account对象。只有在 Account期初余额大于 0 时，才可以开设新帐户。可以在构造器里执行这样的验证。有些人未执行构造器而创建 Account对象，他可能创建了一个具有一些负值的新帐户，这样会使系统不一致，容易受到进一步的干预。

**建议**  
在使用对象之前，请检查对象的初始化过程。要做到这一点，每个类都应该有一个在构造器中设置的私有布尔标志，如清单 9 中的类所示。在每个非 static方法中，代码在任何进一步执行之前都应该检查该标志的值。如果该标志的值为 true，那么控制应该进一步继续；否则，控制应该抛出一个例外并停止执行。那些从构造器调用的方法将不会检查初始化的变量，因为在调用方法时没有设置标志。因为这些方法并不检查标志，所以应该将它们声明为 private以防止用户直接访问它们。

**清单 9. 使用布尔标志以检查初始化过程**

public class MyClass{

private boolean initialized = false;

//Other variables

public MyClass (){

//variable initialization

method1();

initialized = true;

}

private void method1(){ //no need to check for initialization variable

//code

}

public void method2(){

try{

if(initialized==true){

//proceed with the business logic

}

else{

throw new Exception("Illegal State Of the object");

}

}catch(Exception e){

e.printStackTrace();

}

}

}

如果对象由逆序列化进行初始化，那么上面讨论的验证机制将难以奏效，因为在该过程中并不调用构造器。在这种情况下，类应该实现ObjectInputValidation接口：

**清单 10. 实现 ObjectInputValidation**

interface java.io.ObjectInputValidation {

public void validateObject() throws InvalidObjectException;

}

所有验证都应该在 validateObject()方法中执行。对象还必须调用 ObjectInputStream.RegisterValidation()方法以为逆序列化对象之后的验证进行注册。 RegisterValidation()的第一个参数是实现 validateObject()的对象，通常是对对象自身的引用。注：任何实现 validateObject()的对象都可能充当对象验证器，但对象通常验证它自己对其它对象的引用。 RegisterValidation()的第二个参数是一个确定回调顺序的整数优先级，优先级数字大的比优先级数字小的先回调。同一优先级内的回调顺序则不确定。

当对象已逆序列化时， ObjectInputStream按照从高到低的优先级顺序调用每个已注册对象上的 validateObject()。

**不要通过名称来比较类**

有时候，您可能需要比较两个对象的类，以确定它们是否相同；或者，您可能想看看某个对象是否是某个特定类的实例。因为 JVM 可能包括多个具有相同名称的类（具有相同名称但却在不同包内的类），所以您不应该根据名称来比较类。

**影响**  
如果根据名称来比较类，您可能无意中将您不希望授予别人的权利授予了闯入者的类，因为闯入者可以定义与您的类同名的类。

例如，请假设您想确定某个对象是否是类 com.bar.Foo的实例。清单 11 演示了完成这一任务的错误方法：

**清单 11. 比较类的错误方法**

if(obj.getClass().getName().equals("Foo")) // Wrong!

// objects class is named Foo

}else{

// object's class has some other name

}

**建议**  
在那些非得根据名称来比较类的情况下，您必须格外小心，必须确保使用了当前类的 ClassLoader的当前名称空间，如清单 12 中所示：

**清单 12. 比较类的更好方法**

if(obj.getClass() == this.getClassLoader().loadClass("com.bar.Foo")){

// object's class is equal to

//the class that this class calls "com.bar.Foo"

}else{

// object's class is not equal to the class that

// this class calls "com.bar.Foo"

}

然而，比较类的更好方法是直接比较类对象看它们是否相等。例如，如果您想确定两个对象 a和 b是否属同一个类，那么您就应该使用清单 13 中的代码：

**清单 13. 直接比较对象来看它们是否相等**

if(a.getClass() == b.getClass()){

// objects have the same class

}else{

// objects have different classes

}

尽可能少用直接名称比较。

**不要使用内部类**

Java 字节码没有内部类的概念，因为编译器将内部类转换成了普通类，而如果没有将内部类声明为 private，则同一个包内的任何代码恰好能访问该普通类。

**影响**  
因为有这一特性，所以包内的恶意代码可以访问这些内部类。如果内部类能够访问括起外部类的字段，那么情况会变得更糟。可能已经将这些字段声明为 private，这样内部类就被转换成了独立类，但当内部类访问外部类的字段时，编译器就将这些字段从专用（private）的变为在包（package）的作用域内有效的。内部类暴露了已经够糟糕的了，但更糟糕的是编译器使您将某些字段成为 private的举动成为徒劳。

**建议**  
如果能够不使用内部类就不要使用内部类。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#ibm-pcon)

**对付低严重性暴露的技巧**

请遵循下列建议以避免低严重性静态安全性暴露：

* [避免返回可变对象](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#35)
* [检查本机方法](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#38)

**避免返回可变对象**

Java 方法返回对象引用的副本。如果实际对象是可改变的，那么使用这样一个引用调用程序可能会改变它的内容，通常这是我们所不希望见到的。

**影响**  
请考虑这个示例：某个方法返回一个对敏感对象的内部数组的引用，假定该方法的调用程序不改变这些对象。即使数组对象本身是不可改变的，也可以在数组对象以外操作数组的 *内容*，这种操作将反映在返回该数组的对象中。如果该方法返回可改变的对象，那么事情会变得更糟；外部实体可以改变在那个类中声明的 public 变量，这种改变将反映在实际对象中。

清单 14 演示了脆弱性。 getExposedObj()方法返回了 Exposed对象的 *引用副本*，该对象是可变的：

**清单 14. 返回可变对象的引用副本**

class Exposed{

private int id;

private String name;

public Exposed(){

}

public Exposed(int id, String name){

this.id = id;

this.name = name;

}

public int getId(){

return id;

}

public String getName(){

return name;

}

public void setId(int id){

this.id=id;

}

public void setName(String name){

this.name = name;

}

public void display(){

System.out.println("Id = "+ id + " Name = "+ name);

}

}

public class Exp12{

private Exposed exposedObj = new Exposed(1,"Harry Porter");

public Exposed getExposedObj(){

return exposedObj; //returns a reference to the object.

}

public static void main(String[] args){

Exp12 exp12 = new Exp12();

exp12.getExposedObj().display();

Exposed exposed = exp12.getExposedObj();

exposed.setId(10);

exposed.setName("Hacker");

exp12.getExposedObj().display();

}

}

**建议**  
如果方法返回可改变的对象，但又不希望调用程序改变该对象，请修改该方法使之不返回实际对象而是返回它的副本或克隆。要改正清单 14 中的代码，请让它返回 Exposed对象的 *副本*, 如清单 15 中所示：

**清单 15. 返回可变对象的副本**

public Exposed getExposedObj(){

return new Exposed(exposedObj.getId(),exposedObj.getName());

}

或者，您的代码也可以返回 Exposed对象的克隆。

**检查本机方法**

本机方法是一种 Java 方法，其实现是用另一种编程语言编写的，如 C 或 C++。有些开发人员实现本机方法，这是因为 Java 语言即使使用即时（just-in-time）编译器也比许多编译过的语言要慢。其它人需要使用本机代码是为了在 JVM 以外实现特定于平台的功能。

**影响**  
使用本机代码时，请小心，因为对这些代码进行验证是不可能的，而且本机代码可能潜在地允许 applet 绕过通常的安全性管理器（Security Manager）和 Java 对设备访问的控制。

**建议**  
如果非得使用本机方法，那么请检查这些方法以确定：

* 它们返回什么
* 它们获取什么作为参数
* 它们是否绕过安全性检查
* 它们是否是 public、 private等等
* 它们是否含有绕过包边界从而绕过包保护的方法调用

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-staticsec/index.html#ibm-pcon)

**结束语**

编写安全 Java 代码是十分困难的，但本文描述了一些可行的实践来帮您编写安全 Java 代码。这些建议并不能解决您的所有安全性问题，但它们将减少暴露数目。最佳软件安全性实践可以帮助确保软件正常运行。安全至关重要和高可靠系统设计者总是花费大量精力来分析和跟踪软件行为。只有通过将安全性作为至关紧要的系统特性来对待 ― 并且从一开始就将它构建到应用程序中，我们才可以避免亡羊补牢似的、修修补补的安全性方法。

**参考资料**

* 您可以参阅本文在 developerWorks 全球站点上的 [英文原文](http://www.ibm.com/developerworks/library/j-staticsec.html?S_TACT=105AGX52&S_CMP=cn-a-j)。
* 请通过单击文章顶部或底部的 **讨论**来参加本文的 [论坛](javascript:void%20forumWindow())。
* 了解关于 [Java 安全性 API](http://java.sun.com/docs/books/tutorial/security1.2/index.html) 的更多知识。
* *developerWorks*[安全专题](http://www.ibm.com/developerworks/cn/security/index.html)上通常含有有关计算机安全性的优秀资源。
* Larry Koved、Anthony J. Nadalin、Don Neal 和 Tim Lawson 合作编写的“ [The evolution of Java security](http://www.ibm.com/developerworks/cn/java/j-javaevol/index.html)”（ *developerWorks*，1998 年）对 Java 语言的安全性模型早期开发进行了深入探讨。
* Sing Li 在他的 Java 安全性系列文章（由两部分组成的）（ *developerWorks*，2001 年 2 月）中向开发人员显示：尽管社区可能不得不重新考虑 Java 2 中的安全性设计，还是出现了只对开发人员有帮助，可以满足他们的需求的一致的进展：
  + [第一部分](http://www.ibm.com/developerworks/java/library/j-secevol1/index.html?S_TACT=105AGX52&S_CMP=cn-a-j)
  + [第二部分](http://www.ibm.com/developerworks/java/library/j-secevol2/index.html?S_TACT=105AGX52&S_CMP=cn-a-j)
* John Viega、Tom Mutdosch、Gary McGraw 和 Ed Felten 合著的“ [Statically scanning Java code for security vulnerabilities](http://www.viega.org/papers/sw-jss.pdf)”（IEEE Software，2000 年 9 月）介绍了一种 Java 工具，可以使用该工具来检查您的 Java 代码中的安全性漏洞。
* G. McGraw 和 E. Felten 合作编写的 [*Securing Java: Getting Down to Business with Mobile Code*](http://www.amazon.com/exec/obidos/ASIN/047131952X/qid%3D1024501654/104-4457370-3981506)（John Wiley 和 Sons，1998 年）深入涵盖了 Java 安全性。（文档是 PDF 格式的。）
* 定期检查 IBM 研究 [Java 安全](http://www.research.ibm.com/javasec/)页面以便 IBM 在安全性领域的创新有重要发展时能够跟踪这一创新。
* 如果您的 Java 代码运行在 S/390 系统上，那么您将需要查阅 [S/390 Java 安全页面](http://www.ibm.com/servers/eserver/zseries/software/java/security.html)以获取额外的信息。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：这里有数百篇关于 Java 编程各个方面的文章。
* 加入 [developerWorks 中文社区](http://www.ibm.com/developerworks/cn/community/)，查看开发人员推动的博客、论坛、组和维基，并与其他 developerWorks 用户交流。