Java World, 1998 Article

In creating these 12 rules, we've drawn from much experience in hunting down Java security bugs, and on advice and observations from people who write and review security-critical Java code for a living. Each rule is designed to eliminate an unexpected *gotcha* that you might face.

Of course, security is an elusive goal. Following these rules certainly won't provide any guarantee that your code is completely secure. It is easy to write insecure code that follows these rules. By following these goals, however, you will minimize or eliminate certain kinds of security attacks that you might not have thought of.

Think of these rules as a first step. If you are writing code that may be linked or run in conjunction with untrusted code, then you should definitely consider following these rules.

Every attempt has been made to keep the rules simple enough that you can treat them as a checklist to be followed in mechanical fashion. That way you can save your brainpower for other security issues.

**Rule 1: Don't depend on initialization**

Most Java developers think there is no way to allocate an object without running a constructor. But this isn't true: there are several ways to allocate noninitialized objects.

The easy way to protect yourself against this problem is to write your classes so that before any object does anything, it verifies that it has been initialized. You can do this as follows:

* Make all variables private. If you want to allow outside code to access variables in an object, this should be done via get and set methods. (This keeps outside code from accessing noninitialized variables.) If you're following Rule 3, you'll make the get and set methods final.
* Add a new private boolean variable, initialized, to each object.
* Have each constructor set the initialized variable as its last action before returning.
* Have each nonconstructor method verify that initialized is true before doing anything. (Note that you may have to make exceptions to this rule for methods called by your constructors. If you do this, it's best to make the constructors call only private methods.)

If your class has a static initializer, you will need to do the same thing at the class level. Specifically, for any class that has a static initializer, follow these steps:

* + Make all static variables private. If you want to allow outside code to access static variables in the class, this should be done via static get and set methods. This keeps outside code from accessing noninitialized static variables. If you're following Rule 3, you'll make the get and set methods final.
  + Add a new private static boolean variable, classInitialized, to the class.
  + Have the static constructor set the initialized variable as its last action before returning.
  + Before doing anything, have each static method and each constructor verify that classInitialized is true. (Note: constructors are required to call a constructor of the superclass, or another constructor of the same class, as their first action. So you will have to do that before you check classInitialized.)

**Rule 2: Limit access to your classes, methods, and variables**

Every class, method, and variable that is not private provides a potential entry point for an attacker. By default, everything should be private. Make something nonprivate only with good reason, and document that reason.

**Rule 3: Make everything final (unless there's a good reason not to)**

If a class or method isn't final, an attacker could try to extend it in a dangerous and unforeseen way. By default, everything should be final. Make something nonfinal only if there is a good reason, and document that reason.

You might think you can prevent an attacker from extending your class or its methods by declaring the class nonpublic. But if a class isn't public, it must be accessible from within the same package, and as Rule 4 (below) says, you shouldn't to rely on package scope access restrictions for security.

This advice may seem harsh. After all, the rule is asking you to give up extensibility, which is one of the main benefits of using an object-oriented language like Java. But when you're trying to provide security, extensibility is your enemy: it just provides an attacker with more ways to cause trouble.

**Rule 4: Don't depend on package scope**

Classes, methods, and variables that aren't explicitly labeled as public, private, or protected are accessible within the same package. Don't rely on this for security. Java classes aren't closed, so an attacker could introduce a new class into your package and use this new class to access the things you thought you were hiding. (A few packages, such as java.lang, are closed by default, and a few Java virtual machines (JVMs) let you close your own packages. But you're better off assuming packages aren't closed.)

Package scope makes a lot of sense from a software-engineering standpoint, since it prevents innocent, accidental access to things you want to hide. But don't depend on it for security.

Maybe we'll get sealed classes in the future.

**Rule 5: Don't use inner classes**

Some Java language books say inner classes can be accessed only by the outer classes that enclose them. But this isn't true. Java bytecode has no concept of inner classes, so inner classes are translated by the compiler into ordinary classes that happen to be accessible to any code in the same package. And Rule 4 says not to depend on package scope for protection.

But wait, it gets worse. An inner class gets access to the fields of the enclosing outer class, even if the these fields are declared private. And the inner class is translated into a separate class. To let this separate class access the fields of the outer class, the compiler silently changes these fields from private to package scope! It's bad enough that the inner class is exposed; but it's even worse that the compiler is silently overruling your decision to make some fields private. Don't use inner classes if you can help it. (Ironically, the new JDK 1.2 PrivilegedAction API requires you to use an inner class to write privileged code. For more details, see our book *Securing Java* and the *developer.com* article referenced below.) That's one reason we don't like the PrivilegedAction API.)

**Rule 6: Avoid signing your code**

Code that isn't signed will run without any special privileges. And code with no special privileges is much less likely to do damage.

Of course, some of your code might have to acquire and use privileges to perform some dangerous operation. Work hard to minimize the amount of privileged code, and audit the privileged code more carefully than the rest.

**Rule 7: If you must sign your code, put it all in one archive file**

By following this rule, you will help prevent an attacker from carrying out a mix-and-match attack, in which the attacker constructs a new applet or library that links some of your signed classes together with malicious classes, or links together signed classes that you never meant to be used together. By signing a group of classes together, you make such attacks more difficult. Existing code-signing systems do an inadequate job of preventing mix-and-match attacks, so this rule cannot prevent such attacks completely. But using a single archive can't hurt.

Some code-signing systems let you examine other classes to see who signed them. If you're using a code-signing system that allows this, you can put code into the static constructors of your classes to verify that the "surrounding" classes have been signed by the expected person.

This measure doesn't completely prevent mix-and-match attacks, since an adversary can still mix together classes you signed at different times -- for example, by mixing version 1 of Class A with version 2 of Class B. If you're worried about this kind of inter-version mix-and-match attack, you can put each class's "version stamp" in a public final variable, and then have each class check the version stamps of its surrounding classes.

**Rule 8: Make your classes noncloneable**

Java's object cloning mechanism can allow an attacker to manufacture new instances of classes you define, without executing any of your constructors. If your class isn't cloneable, the attacker can define a subclass of your class, and make the subclass implement java.lang.Cloneable. This lets an attacker create new instances of your class. The new instances are made by copying the memory images of existing objects; though this is sometimes an acceptable way to make a new object, it often is not.

Rather than worry about this, you're better off making your objects noncloneable. You can do this by defining the following method in each of your classes:

**public** **final** **void** clone() **throws** java.lang.**CloneNotSupportedException** {

**throw** **new** java.lang.**CloneNotSupportedException**();

}

**If** you want your **class** to be cloneable, **and** you've considered the

consequences of that choice, then you can still protect yourself. If you're

defining a clone method yourself, make it **final**. **If** you're relying on a

nonfinal clone method in one of your superclasses, then define this

method:

public final void clone() throws java.lang.CloneNotSupportedException {

super.clone();

}

This prevents an attacker from redefining your clone method.

**Rule 9: Make your classes nonserializeable**

Serialization is dangerous because it allows adversaries to get their hands on the internal state of your objects. An adversary can serialize one of your objects into a byte array that can be read. This allows the adversary to inspect the full internal state of your object, including any fields you marked private, and including the internal state of any objects you reference.

To prevent this, you can make your object impossible to serialize. To achieve this goal, declare the writeObject method:

**private** **final** **void** writeObject(**ObjectOutputStream** **out**)

**throws** java.io.**IOException** {

**throw** **new** java.io.**IOException**("Object cannot be serialized");

}

This method is declared final so that a subclass defined by the adversary cannot override it.

**Rule 10: Make your classes nondeserializeable**

This rule is even more important than the previous one. Even if your class isn't serializeable, it may still be deserializeable. An adversary can create a sequence of bytes that happens to deserialize to an instance of your class. This is dangerous, since you do not have control over what state the deserialized object is in. You can think of deserialization as another kind of public constructor for your object; unfortunately it's a kind of constructor that is difficult for you to control.

You can prevent this kind of attack by making it impossible to deserialize a byte stream into an instance of your class. You can do this by declaring the readObject method:

**private** **final** **void** readObject(**ObjectInputStream** **in**)

**throws** java.io.**IOException** {

**throw** **new** java.io.**IOException**("Class cannot be deserialized");

}

As above, this method is declared final to prevent the adversary from overriding it.

**Rule 11: Don't compare classes by name**

Sometimes you want to compare the classes of two objects to see whether they are the same; or you want to see whether an object has a particular class. When you do this, be aware that there can be multiple classes with the same name in a JVM. It is a mistake to compare classes by name since different classes can have the same name. A better method is to compare class objects for equality directly. For example, given two objects, A and B, if you want to see whether they are the same class, use this code:

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

*// objects have the same class*

}**else**{

*// objects have different classes*

}

**You** should also be on the lookout **for** cases of less direct **by**-name

comparisons. **Suppose**, **for** example, you want to see whether an **object**

has the **class** "Foo." **Here** **is** the wrong way to **do** it:

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

*// objects class is named Foo*

}**else**{

*// object's class has some other name*

}

**Here**'s a better way to do it:

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

// object's **class** **is** equal to the **class** that **this** **class** calls "Foo"

}**else**{

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

*// this class calls "Foo"*

}

Do note the legalistic comments in the last example. Whenever you use class names, you open yourself up to mix-and-match attacks, as described in Rule 7. You should also know that the Java language forces you to use class names all the time: in variable declarations, instanceof expressions, and exception-catching blocks. Only the designers of Java can prevent mix-and-match attacks, but you can avoid making the problem worse by avoiding by-name class comparisons.

## Rule 12: Secrets stored in your code won't protect you

You might be tempted to store secrets such as cryptographic keys in the code for your application or library. Secrets stored in this way are completely accessible to anybody who runs your code. There is nothing to stop a malicious programmer or virtual machine from looking inside your code and learning its secrets.

Code obfuscation is another way of storing a secret in your code; in the case of obfuscation the secret is simply the algorithm used by your code. There's not much harm in using an obfuscator, but you shouldn't believe it will provide strong protection. There is no real evidence that it is possible to obfuscate Java source code or bytecode so that a dedicated adversary with good tools cannot reverse the obfuscation.

## Conclusion

Writing secure Java code is very difficult. There is no magic bullet that will solve your security problems; all you can do is think hard (perhaps with help from formal analysis tools) and use prudent engineering practices to minimize risks. Sometimes a pair of objective outside eyes can help. The rules set forth here are intended to describe prudent engineering practices for writing secure Java code. They won't solve your security problems, but they will reduce the number of ways things can go wrong.

The small print: Portions of this article are taken by permission from Securing Java: Getting down to business with mobile code (John Wiley & Sons, 1998), the second edition of McGraw and Felten's groundbreaking book Java Security: Hostile Applets, Holes, and Antidotes (Wiley, 1996)

Gary McGraw, PhD, vice president at Reliable Software Technologies, is a noted authority on Java security who has written over fifty peer-reviewed technical publications, and consults with major e-commerce vendors including Visa. Edward Felten is an assistant professor of computer science at Princeton, where he leads Princeton's Secure Internet Programming Team.