Java Insecurity: Accounting for Subtleties That Can Compromise Code

Charlie Lai, *Sun Microsystems*

Subtleties related to fundamental Java coding guidelines can lead to unexpected behavior and security vulnerabilities. Learn how to safely account for these subtleties to prevent attacks.

Conscientious Java developers are typically aware of the numerous coding guidelines that they should follow when writing code, such as validating inputs, minimizing accessibility to classes and members, and avoiding public static nonfinal fields. Java developers follow such guidelines to avoid common programming pitfalls (often called antipatterns), thereby reducing the likelihood of bugs or security vulnerabilities in their programs.

These example guidelines are fundamental. Adhering to them can make any Java program more robust. They also help establish a basic foundation on which to build higher-level guidelines that focus on application-specific threats, such as cross-site scripting and SQL-injection attacks. Together, the foundational and higher-level guidelines help shield a program from varying levels of attack and from its own internal errors.

Many of the fundamental guidelines are also simple to follow. It doesn't take much thought to determine whether a method should be declared public or private. However, a seemingly simple guideline often has associated subtleties that (if not accounted for) can undermine the very security that the guideline was intended to provide. For example, although a developer might declare a class member private to minimize its accessibility, the Java compiler, under certain circumstances, can widen the accessibility from private to package-private. Such a subtlety not only unravels the developer's intent but also can break his or her fundamental assumptions and place the entire program's security at risk.

Here, I examine numerous subtleties associated with three fundamental coding guidelines—minimizing accessibility, creating copies of mutable objects, and preventing the unauthorized construction of sensitive classes. I also discuss the potential threats that these subtleties pose and describe the appropriate defensive measures Java developers can apply to counter them. By understanding and accounting for these subtleties, developers can ensure that their programs' underlying security remains sound and uncompromised.

Private property

One of the first decisions a Java developer faces when designing a new class or member is determining its accessibility. From a security perspective, developers should minimize accessibility as much as possible.[1] For example, if a class member is part of neither a published specification (a public API) nor the package implementation, it should be declared private. This defends against direct attacks from outside the package and unintended use from within a package.

0740-7459/08/$25.00 © 2008 IEEE

```
package mypackage;

public final class EnclosingClass {

    // private nested class
    private static final class NestedClass {
        private int nestedInt = 10;
    }

    public static void main(String[] args) {
        EnclosingClass.NestedClass nested = new EnclosingClass.NestedClass();
        System.out.println("nestedInt = " + nested.nestedInt);
    }
}
```

Figure 1. An example nested class.

```
%javac EnclosingClass.java

%javap -private EnclosingClass.NestedClass
Compiled from "EnclosingClass.java"
final class mypackage.EnclosingClass$NestedClass extends java.lang.Object {
    private int nestedInt;
    static int access$100(EnclosingClass$NestedClass);
     ...
}
```

Figure 2. Introspecting class files with javap.

As I mentioned earlier, however, a private declaration doesn't always remain private. Under certain circumstances, the Java compiler automatically widens the accessibility of a class member from private to package-private without any notification. Such a fundamental change to a class can break assumptions the developer made and affect higher-level layers of security. So, it's important to understand specifically when this can occur and exactly what the risk is.

The automatic widening of accessibility from private to package-private occurs solely with the use of nested classes (see figure 1).

Java's rules let an enclosing class access private members inside a nested class and vice versa. In the example in figure 1, the `main` method in `EnclosingClass` is permitted to create an instance of the private class, `NestedClass`, and may also access the internal private field, `nestedInt`.

The compiler does enforce the proper access controls at compile time. Attempts to access `NestedClass` from classes other than `EnclosingClass` result in a compiler error. However, the class files that the compiler generates result in a different set of access controls being enforced at runtime. To help understand what causes this change, use the `javap` tool to introspect those class files (see figure 2).

The output from `javap` doesn't show the private modifier for `NestedClass`. The class has been converted to package-private. In addition, the output lists a new static method, `access$100`, which the developer didn't define. The compiler automatically generated this method, which is called a *synthetic method*. The implementation of `access$100` returns the value stored in the private `nestedInt` field. Because the method itself is package-private, it effectively gives package-wide access to the private field. In short, both `NestedClass` and `nestedInt` have been converted from private to package-private.

Recall that the compiler enforces the proper language-access controls at compile time, so there's no threat of trusted classes unintentionally accessing the newly designated package-private members. However, the members (in fact, any package-private member) can come under attack if hostile code can infiltrate the package. A package attack involves adding new classes to a package, replacing existing classes in a package, or both.

Such attacks are possible if trusted code is hosted in a virtual machine alongside untrusted code. This scenario is best exemplified in a user's Web browser, where the Java plug-in can load applets from unrelated sources into the same VM. The key to preventing package attacks in such untrusted environments is to properly isolate code using class loaders.

Class loaders are responsible for defining packages into the Java VM. As a result, packages are scoped not only by their name but also by the class loader that defined them. Even if different class loader instances defined packages with the same name, the Java VM would treat them as completely separate packages. This means that for a successful package attack to occur, hostile code must be loaded into the specific class loader instance that defined the target package.

To ensure that such an attack isn't possible, services that perform class loading must isolate unrelated code into separate class loader instances. The Java plug-in, for example, loads unrelated applets into separate class loader instances. This isolates hostile applets from other applets, even if they happen to share the same package name. In fact, class loaders can also be used to isolate unrelated units of trusted code. For example, different Web applications deployed in an application server container shouldn't have access to each other's respective packages, even by accident. So, Web containers isolate trusted Web applications using separate class loaders as well.

Using private nested classes isn't at all discouraged; on the contrary, private nested classes can often simplify the code's readability and maintainability. The original decision to support private nested classes in the current manner (by widening the scope of certain private members to package-private) reflects a trade-off made early in Java's history. Alternative solutions would have required changes to the class file format, thereby limiting newly written programs' write-once-run-anywhere capability. To ensure maximum compatibility and to encourage Java's adoption, private nested classes were designed in a way that didn't require such changes.

Copy that

Another fundamental software-coding guideline involves the proper validation of inputs. If the input object is mutable, it's equally important to make a copy of that object before validating it. Otherwise, the caller who passed the input can mutate it to exploit race conditions in the method. A typical exploitable race condition is a "time-of-check, time-of-use" inconsistency,[2] where a mutable input contains one value during a validation check but a different value when it's used later.

The public entry points of a shared library represent the most noteworthy places to make copies of mutable inputs because library code typically can't trust its callers. Although such copies are primarily meant to defend against hostile callers maliciously tampering with inputs, they can also serve to protect against fully trusted callers inadvertently modifying inputs.

In Java, the simplest way to create a copy of a mutable object is via its cloning mechanism. If the object is an instance of `java.lang.Cloneable` and provides a public `clone` method, then callers can invoke that method to create a copy of the object. The example in figure 3 demonstrates the use of `clone`.

Java developers must be aware of two issues related to `clone`. First, not all classes implement `Cloneable`. Second, even if a class does implement `Cloneable`, invoking its `clone` method might not be appropriate (safe). In the example in figure 3, it's only safe to invoke `clone` on `HttpCookie` because that class is declared final. If the class were nonfinal, an untrusted caller could subclass it, maliciously override the `clone` implementation, and then pass instances of the subclass to the method.

The basic workaround in these two cases is to create a manual copy of the mutable object. Retrieve all the state from the original object, and then create a new instance of that object with the original state (see figure 4).

```
public void cloneIt(int[] inputInts, java.net.HttpCookie inputCookie) {
    // arrays of primitives can be cloned
    inputInts = inputInts.clone();

    // HttpCookie is mutable and implements Cloneable
    inputCookie = inputCookie.clone();

    // continue
}
```

Figure 3. Cloning inputs.

```
public void manualCopy(java.lang.StringBuilder builder) {
    // StringBuilder is mutable, but does not implement Cloneable

    // first get original state
    String originalState = builder.toString();

    // then create new instance with that state
    builder = new StringBuilder(originalState);

    // continue
}
```

Figure 4. Creating a manual copy of a mutable object.

The manual copy procedure can become complicated if the retrieved state itself is mutable. To achieve a deep copy of the original object, such state must also be (deep) copied.

Although manual copies can be appropriate, they can also be extremely fragile. If the class whose instances are being copied (`StringBuilder` in this case) evolves to include additional state in the future, then the copy procedure used in `manualCopy` might become stale—the additional state might be left out of its copies. For the manual copy procedure to remain correct, it must evolve in tandem with the class that's being copied. Specifically, it must ensure that any additional state in `StringBuilder` is likewise retrieved and passed to the new instance.

Often, however, the input type and the method itself aren't under the same developer's control. Such is the case in the example in figure 4. The danger is that `manualCopy`'s developer might not notice when changes occur to `StringBuilder`. Ironically, this is especially true if `StringBuilder` evolves in a compatible fashion. In that case, the original `manualCopy` implementation will continue to compile and execute as if nothing changed. The missing additional state will likely be assigned default values in the newly created copies. Although this ensures that the original copy procedure will continue to execute, it doesn't ensure that it will execute securely.

In this particular case, it's unlikely that `StringBuilder` will evolve in a way that will break `manualCopy`.

```
public void copyNonFinal(java.util.ArrayList<String> inputList) {
    // ArrayList is mutable and not declared final.
    // It also implements Cloneable.
    //
    // - Invoke clone method?
    //          inputList = inputList.clone();
    //
    // - Create a manual copy?
    //          inputList = new ArrayList<String>(inputList);
}
```

Figure 5. Copying a nonfinal mutable input.

```
public void copyInterface(java.util.Collection inputCollection) {
   // java.util.Collection is an interface
   // - create instance of trusted standard Collection implementation
   inputCollection = new java.util.ArrayList(inputCollection);

   // continue
}
```

Figure 6. Converting a mutable input into a trusted type.

```
public final class Sensitive {
    // sole constructor
    public Sensitive() {
        // first perform security check
        securityManagerCheck();

        // remainder of constructor implementation
    }

    // custom security check
    private void securityManagerCheck() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            // security exception thrown if check fails
            sm.checkPermission(...);
        }
    }

    // remainder of class
}
```

Figure 7. Guarding instantiation with security checks.

Nevertheless, developers must apply manual copies with care and not simply as a matter of fact to work around specific issues related to `clone`. If the developer deems a manual copy appropriate, then it's important to closely monitor design changes to the type being copied.

Moreover, in the previous example, it's safe to create a manual copy of `StringBuilder` only because `StringBuilder` is declared final. Figure 5 shows another example where the input type isn't final.

As I discussed earlier, invoking `clone` on the nonfinal `ArrayList` might not be safe because untrusted callers can override that method. A manual copy might be appropriate, but only with certain restrictions. In the example in figure 5, consider if the input object were an instance of a subclass of `ArrayList`. Because the manual copy procedure specifically creates a new instance of `ArrayList` itself, it would discard any subclass and corresponding subclass state from the resulting copy. In other words, the resulting copy wouldn't be identical to the original input object.

In a related example, if a mutable input type were an interface, a method might perform a manual conversion of the input into a trusted implementation (see figure 6).

With manual conversion, the resulting copy is also not guaranteed to be identical to the original input object. In this case, the input is potentially converted into an entirely different `Collection` type.

Although a nonidentical copy of an input is far from ideal, it can be acceptable. Methods whose input types are nonfinal (as in the case of `copyNonFinal`) or are interfaces (as in the case of `copyInterface`) typically have no dependencies on specific subtypes. Such methods have dependencies only on the declared type itself, so discarding or converting a subtype from an input object likely doesn't pose any problem.

However, such copies must not be passed to other objects or returned to the caller. Although a discarded or converted subtype might not affect the method creating the copy, it can affect downstream code. Such code can attempt to cast the object back into specific subtypes. If an underlying subtype changes or disappears, overall program behavior can be altered, adversely affecting security.

Clearly, mutable (including nonfinal) inputs place a significant burden on public method implementations. To help minimize this burden, favor immutability when designing new classes.[1] When designing a mutable class, provide a means to create a copy of it. You can do this through a copy constructor or, if the class is final, by implementing `Cloneable` and declaring a public `clone` method.

Under construction

Most public classes are meant to be freely instantiated. Those that aren't tend to be powerful classes, often sensitive to security. Consider the public class `java.lang.ClassLoader`. Class loaders are central to the Java VM's security architecture and have the power to define classes with arbitrary security permissions. So, the construction of `ClassLoader` instances is restricted.

The most straightforward way to prevent the unauthorized construction of a security-sensitive class, such as ClassLoader, is to enforce a java.lang.SecurityManager check in all its public and protected constructors (see figure 7). Chapter 6 of *Inside Java 2 Platform Security* gives more information on implementing custom SecurityManager checks.[3]

java.lang.ClassLoader enforces the same kind of SecurityManager checks in all its accessible constructors. With the check in place, any unauthorized attempt to directly instantiate the class results in a java.lang.SecurityException being thrown:

```
// unauthorized calls result in a java.lang.SecurityException
Sensitive authorizedInstance = new Sensitive();
```

Often overlooked, however, is that in Java, acquiring instances of a class without invoking new is possible. This can occur in two scenarios: when a class is cloneable (it implements java.lang.Cloneable) and when a class is serializable (it implements java.io.Serializable). In the case of cloning, a caller can acquire an instance (copy) of a cloneable class by invoking the public clone method, as I described earlier. In the case of deserialization, an instance of a serializable class can be reconstituted from a serialized stream (see figure 8).

These two scenarios are noteworthy because they produce instances of a class without invoking any constructor in that class. In the case of cloning, a new instance (copy) of the class is produced via a custom clone method implementation. Although a clone implementation could theoretically invoke a constructor in the class and return the newly created instance, this behavior is discouraged. A proper implementation of clone returns the instance obtained via super.clone, after performing any necessary fixes.[1] The call to super.clone eventually terminates in java.lang.Object.clone, which creates the actual clone instance without invoking any constructor in the original class.

When deserializing a class that implements java.io.Serializable, Java's serialization framework (which includes the implementation of ObjectInputStream) produces a new instance of that class without invoking any of its constructors.

Whether a constructor is invoked is important because the security check that prevents unauthorized instantiation resides in the constructor. If a cloneable or serializable class doesn't account for this, the check can be trivially bypassed. Specifically, the security check enforced in a class constructor must be duplicated in the clone method of a cloneable class and in the readObject (or readObjectNoData) method of a serializable class (see figure 9).

```
// serialized object resides in file
java.io.FileInputStream fis = new java.io.FileInputStream("serializedObjectFile");
java.io.ObjectInputStream ois = new java.io.ObjectInputStream(fis);

// reconstitute instance of serialized object from file stream
SerializableClass myInstance = (SerializableClass)ois.readObject();
```

Figure 8. Object deserialization.

```
public final class Sensitive implements java.lang.Cloneable, java.io.Serializable {
    // sole constructor
    public Sensitive() {
        // first perform security check
        securityManagerCheck();

        // remainder of constructor implementation
    }

    public Object clone() throws java.lang.CloneNotSupportedException {
        // duplicate constructor security check
        securityManagerCheck();

        // remainder of clone implementation
    }

    private void readObject(java.io.ObjectInputStream) throws java.io.IOException {
        // duplicate constructor security check
        securityManagerCheck();

        // remainder of readObject implementation
    }

    private void securityManagerCheck() { ... }

    // remainder of class
}
```

Figure 9. Duplicating a constructor security check in clone and readObject.

The class in figure 9 properly defends against unauthorized instantiation. However, the coded solution is only secure because the class is declared final. If the class weren't declared final, a partially initialized instance of it could still be leaked during object finalization.

When an object is no longer referenced in Java, an internal garbage collector automatically reclaims that object's memory. As part of this process, the finalize method (either inherited from java.lang.Object or overridden in a subclass) is invoked on that object. The example in figure 10 demonstrates how a partially initialized instance can be exposed in an overridden finalize implementation.

NonFinalSensitive isn't declared final. It has a protected method, run, accessible to and overridable by subclasses. The class defends against unauthorized

```
public class NonFinalSensitive {
    // sole constructor which potentially throws runtime SecurityException
    public NonFinalSensitive() {
        securityManagerCheck();
        init();
    }

    // method accessible to subclasses
    protected void run() {
        // do something
    }
}

// extend nonfinal class
public class Tricky extends NonFinalSensitive {
    public static void main(String[] args) {
        try {
            new Tricky();
        } catch (SecurityException se) {
            // handle exception
        }
    }

    protected void finalize() throws Throwable {
        // Java VM garbage collector invoking cleanup on this (partially initialized) object
        NonFinalSensitive partiallyInitialized = this;
        partiallyInitialized.run();
    }
}
```

Figure 10. Partially initialized instances and finalize.

instantiation with a constructor security check. If this check fails, the constructor throws a SecurityException. Because NonFinalSensitive implements neither Serializable nor Cloneable, it doesn't need to duplicate the constructor check.

Tricky extends NonFinalSensitive and overrides the finalize method inherited from java.lang.Object. The main method in Tricky first attempts to create an instance of the class. The default constructor implicitly calls super, which calls into the no-argument constructor for NonFinalSensitive. NonFinalSensitive's security check properly blocks instantiation and throws a SecurityException. The main method in Tricky handles the exception, however. When the Java VM performs garbage collection on the partially initialized object, Tricky's finalize method is invoked, where it has access to this, a reference to the partially initialized object being finalized. Tricky can then invoke methods on that object. The original constructor security check inside NonFinalSensitive has inadvertently been bypassed.

The leak of partially initialized instances during finalization isn't merely restricted to cases where SecurityException is thrown. Any type of exception thrown from the constructor of a nonfinal class can cause the leak to occur. Moreover, if NonFinalSensitive were hosted in a VM alongside untrusted code, the problem would be greatly exacerbated. In such cases, an attacker can directly implement a class similar to Tricky with malicious intent. This is commonly called a *finalizer attack*.

Finalizers have issues other than simply the leak of partially initialized instances. They're fundamentally problematic—to the point where their use is generally discouraged.[1] So, the most straightforward way to prevent partially initialized instances from becoming a problem is to avoid using finalizers altogether. In untrusted environments, the finalize method can simply be declared final.

In the rare case where finalization must be securely accommodated, use an initialized flag (see figure 11).

The initialized flag is set as the last step in a constructor before returning successfully. Each accessible method in the class must first consult the flag before performing any logic. If the flag isn't set, an exception must be thrown. This solution doesn't prevent partially initialized instances from being acquired, but it does prevent such instances from doing anything useful for a potential attacker.

Admittedly, the flag isn't aesthetically pleasing. Its purpose isn't immediately obvious to developers reading the source code. Moreover, the flag intrudes throughout the class—it appears in all accessible methods. A developer should consider the potential for mischief before deciding whether an initialized flag is appropriate.

If an object is only partially initialized, its internal fields likely contain safe default values such as null. Even in an untrusted environment, such an object is unlikely to be useful to an attacker. If the developer deems the partially initialized object state secure, then the developer doesn't have to pollute the class with the flag. The flag is necessary only when such a state isn't secure or when accessible methods in the class perform sensitive operations without referencing any internal field.

In this article, I've raised some questions that Java developers can ask themselves during development to help draw more attention to subtleties that can leave their programs at risk. How is code affected if a class is mutable? If the class isn't declared final? If it implements Cloneable? If it implements Serializable? Is the coded solution safe in the presence of evolving classes? Remembering to answer such questions can help developers detect problems that might not be obvious at first glance

and enable them to take the appropriate defensive measures to properly secure their code.

Developers can apply these defensive techniques to any Java program, whether deployed in a trusted or untrusted environment. Although hostile exploits tend to garner the most attention, internal errors in fully trusted environments can be equally crippling. To reinforce the latter point, consider this quote in the *New York Times* from Peter G. Neumann, a security expert and principal scientist at SRI International: "We don't need hackers to break the systems because they're falling apart by themselves."[4]

To ensure that code remains as robust as possible, supplement visual code reviews with the use of static analysis tools such as FindBugs.[5] These tools can help ease identifying the subtleties described in this article and can also help autodetect when common coding guidelines aren't followed at all.

```
public class NonFinalSensitive {
    private volatile boolean initialized = false;

    // sole constructor which potentially throws run-time SecurityException
    public NonFinalSensitive() {
        securityManagerCheck();
        init();
        // set flag before returning
        initialized = true;
    }

    // method accessible to subclasses
    protected void run() {
        // check flag as first step
        if (!initialized) {  throwException(); }
        // do something
    }
}
```

Figure 11. Using an initialized flag to thwart attacks against partially initialized objects.

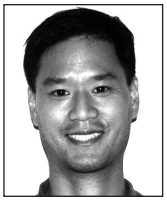
Acknowledgments

I thank Tom Hawtin, Jeff Nisewanger, Valerie Peng, Bob Scheifler, and Andreas Sterbenz for their input on the ideas presented in this article.

References

1. J. Bloch, *Effective Java Programming Language Guide*, 1st ed., Addison-Wesley, 2001.
2. G. McGraw, *Software Security: Building Security In*, Addison-Wesley, 2006.
3. L. Gong, G. Ellison, and M. Dageforde, *Inside Java 2 Platform Security*, 2nd ed., Addison-Wesley, 2003.
4. J. Schwartz, "Who Needs Hackers?" *New York Times*, 12 Sept. 2007.
5. D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, 2004, pp. 92–106.

About the Author



Charlie Lai is part of Sun Microsystems' Java security team, where he's worked on integrating user authentication capabilities into the security architecture of the Java Development Kit and creating Java key store that enables access to cryptographic keys and certificates stored on PKCS#11 tokens. Before joining the Java security team, Charlie worked in Sun's Solaris security group, where he implemented a pluggable authentication framework. He received his master's in computer science from the University of Southern California. Contact him at SCA22-226, 4220 Network Circle, Santa Clara, CA 95054; charlie.lai@sun.com.