Secure Coding Guidelines for the Java Programming Language, version 2.0

Introduction

It has been noted that writing correct software is a problem beyond the ability of computer science to solve [1]. Although large-scale software projects are most often thought to be fraught with defects, even modest, well-tested programs can contain bugs that lead to significant security vulnerabilities. Security holes are all too common in software, and the problem is only growing [2].

The choice of programming language can impact the robustness of a software program. The Java language [3] and virtual machine [4] provide many features to help developers avoid common programming mistakes. The language is type-safe, and the runtime provides automatic memory management and range-checking on arrays. These features also make Java programs immune to the stack-smashing [5] and buffer overflow attacks possible in the C and C++ programming languages, and that have been described as the single most pernicious problem in computer security today [6].

On the flip side, the Java platform has its own unique set of security challenges. One of its main design considerations is to provide a secure environment for executing mobile code. While the Java security architecture [7] can protect users and systems from hostile programs downloaded over a network, it can not defend against implementation bugs that occur in *trusted* programs. Such bugs can inadvertently open the very holes that the security architecture was designed to contain, including the leak of private information, the abuse of privileges, and ultimately the access of sensitive resources by unauthorized users.

To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should adhere to recommended coding guidelines. Existing publications, including [8], provide excellent guidelines related to Java software design. Others, including [6], outline guiding principles for software security. This paper bridges such publications together, and includes coverage of additional topics to provide a more complete set of security-specific coding guidelines targeted at the Java programming language. The guidelines are of interest to all Java developers, whether they implement the internals of a security component, develop shared Java class libraries that perform common programming tasks, or create end user applications. Any implementation bug can have serious security ramifications, and can appear in any layer of the software stack.

1 Accessibility and Extensibility

Guideline 1-1 Limit the accessibility of classes, interfaces, methods, and fields

A Java package comprises a grouping of related Java classes and interfaces. Declare any class or interface public if it is specified as part of a published application programming interface (API). Otherwise, declare it package-private. Likewise, declare all respective class members (nested classes, methods, or fields) public or protected as appropriate, if they are also part of the API. Otherwise, declare them package-private if they are part of the package implementation, or private if they exist solely as part of a class implementation.

In addition, refrain from increasing the accessibility of an inherited method, as doing so may break assumptions made by the superclass. A class that overrides the protected `java.lang.Object.finalize` method and declares that method public, for example, enables hostile callers to finalize an instance of that class, and to call methods on that instance after it has been finalized. A superclass implementation unprepared to handle such a call sequence could throw runtime exceptions that leak private information, or that leave the object in an invalid state that compromises security. One noteworthy exception to this guideline pertains to classes that implement the `java.lang.Cloneable` interface. In these cases, the accessibility of the `Object.clone` method should be increased from protected to public (see the javadoc for Cloneable and Guideline 2-2).

Also note that the use of nested classes can automatically cause the accessibility of members in both the nested class and its enclosing class to widen from private to package-private. This occurs because the javac compiler adds new static package-private methods to the generated class file to give nested classes direct access to referenced private members in the enclosing class and vice versa. Any nested class declared private is also converted to package-private by the compiler. While javac disallows the new package-private methods from being called at compile-time, the methods - in fact, any protected or package-private class or member - can be exploited at run-time using a package insertion attack (an attack where hostile code declares itself to be in the same package as the target code). In the presence of nested classes, this attack is particularly pernicious because it gives the attacker access to class members originally declared private by a developer.

Package insertion attacks can be difficult to achieve in practice. In the Java virtual machine, class loaders are responsible for defining packages. For a successful attack to occur, hostile code must be loaded by same class loader instance as the target code. As long as services that perform class loading properly isolate unrelated code (the Java Plugin, for example, loads unrelated applets into separate class loader instances), untrusted code can not access package-private members declared in other classes, even if it declares itself to be in the same package.

Guideline 1-2 Limit the extensibility of classes and methods

Design classes and methods for inheritance, or else declare them final [8]. Left non-final, a class or method can be maliciously overridden by an attacker.

If a class is public and non-final, and wants to limit subclassing solely to trusted implementations, confirm the class type of the instance being created. This must be done at all points where an instance of the non-final class can be created (see Guideline 4-1). If a subclass is detected, enforce a `SecurityManager` check (see Chaper 6 of [7]) to block malicious implementations:

```
public class NonFinal {

    // sole constructor
    public NonFinal() {

        // invoke java.lang.Object.getClass to get class instance
        Class clazz = getClass();

        // confirm class type
        if (clazz != NonFinal.class) {

            // permission needed to subclass NonFinal
            securityManagerCheck();

        }
        // continue
    }

    private void securityManagerCheck() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(...);
        }
    }

}
```

Confirm an object's class type by examining the java.lang.Class instance belonging to that object. Do not compare Class instances solely using class names (acquired via `Class.getName`), since instances are scoped both by their class name as well as the class loader that defined the class.

Guideline 1-3 Understand how a superclass can affect subclass behavior

Subclasses do not have the ability to maintain absolute control over their own behavior. A superclass can affect subclass behavior by changing the implementation of an inherited method not overridden. If a subclass overrides all inherited methods, a superclass can still affect subclass behavior by introducing new methods. Such changes to a superclass can unintentionally break assumptions made in a subclass and lead to subtle security vulnerabilities. Consider the following example that occurred in JDK 1.2:

```
        Class Hierarchy                    Inherited Methods
        -----------------------            --------------------------
          java.util.Hashtable              put(key, val)
                    ^                      remove(key)
                    | extends
                    |
          java.util.Properties
                    ^
                    | extends
                    |
         java.security.Provider            put(key, val) //
SecurityManager put check
                                           remove(key)   //
SecurityManager remove check
```

`java.security.Provider` extends from java.util.Properties, and Properties extends from java.util.Hashtable. In this hierarchy, Provider inherits certain methods from Hashtable, including *put* and *remove*. *Provider.put* maps a cryptographic algorithm name, like RSA, to a class that implements that algorithm. To prevent malicious code from affecting its internal mappings, Provider overrides *put* and *remove* to enforce the necessary SecurityManager checks.

The Hashtable class was enhanced in JDK 1.2 to include a new method, `entrySet`, which supports the removal of entries from the Hashtable. The Provider class was not updated to override this new method. This oversight allowed an attacker to bypass the SecurityManager check enforced in `Provider.remove`, and to delete Provider mappings by simply invoking the `Hashtable.entrySet` method.

The primary flaw is that the data belonging to Provider (its mappings) are stored in the Hashtable class, whereas the checks that guard the data are enforced in the Provider class. This separation of data from its corresponding SecurityManager checks only exists because Provider extends from Hashtable. Because a Provider is not inherently a Hashtable, it should not extend from Hashtable. Instead, the Provider class should encapsulate a Hashtable instance, allowing the data and the checks that guard that data to reside in the same class. The original decision to subclass Hashtable likely resulted from an attempt to achieve code reuse, but it unfortunately led to an awkward relationship between a superclass and its subclasses, and eventually to a security vulnerability.

2 Input and Output Parameters

Guideline 2-1 Create a copy of mutable inputs and outputs

If a method is not specified to operate directly on a mutable input parameter, then create a copy of that input and only perform method logic on the copy. Otherwise, a hostile caller can modify the input to exploit race conditions in the method. In fact, if the input is stored in a field, the caller can exploit race conditions in the enclosing class. For example, a "time-of-check, time-of-use" inconsistency (TOCTOU) [2] can be exploited, where a mutable input contains one value during a SecurityManager check, but a different value when the input is later used.

```
public final class Copy {

    // java.net.HttpCookie is mutable
    public void copyMutableInput(HttpCookie cookie) {
        if (cookie == null) {
            throw new NullPointerException();
        }

        // create copy
        cookie = cookie.clone();

        // perform logic (including relevant security checks) on copy
        doLogic(cookie);
    }
}
```

To create a copy of a mutable object, invoke an appropriate method on that object (see Guideline 2-2). HttpCookie is final and provides a public *clone* method for acquiring copies of its instances.

If the input type is non-final, the `clone` method may be overridden in a malicious subclass. Ideally a non-final input defends against this by blocking malicious subclassing (see Guideline 1-2). Without a source code review, however, a receiving method implementation can not confirm this. Also, many classes do not defend against malicious subclassing, and have no obvious reason to do so. This is true of the standard collections, `java.util.ArrayList` and `java.util.HashSet`.

Under certain circumstances, the following approaches can be used to overcome the difficulty of copying a mutable input whose type is non-final, or is an interface. If the input type is non-final, create a new instance of that non-final type:

```
    // java.util.ArrayList is mutable and non-final
    public void copyNonFinalInput(ArrayList list) {
        // create new instance of declared input type
        list = new ArrayList(list);
        doLogic(list);
    }
```

If the input type is an interface, create a new instance of a trusted interface implementation:

```
    // java.util.Collection is an interface
    public void copyInterfaceInput(Collection collection) {
        // convert input to trusted implementation
        collection = new ArrayList(collection);
        doLogic(collection);
    }
```

Neither approach produces a copy that is guaranteed to be identical to the original input. Creating a new instance of a non-final input discards any potential subclass information. Creating a new instance of a trusted collection implementation potentially converts the input collection type into an entirely different collection type. Such approaches can be safe to use, however, if the method performing the copy only relies on the behavior defined by the declared input type, and if the produced copy is not passed to other objects.

In some cases, a method may require a deeper copy of an input object than the one returned via that input's copy constructor or *clone* method. Invoking *clone* on an array, for example, produces a shallow copy of the original array instance. Both the copy and the original share references to the same elements. If a method requires a deep copy over the elements, it must create those copies manually:

```
public void deepCopy(int[] ints, HttpCookie[] cookies) {
    if (ints == null || cookies == null) {
        throw new NullPointerException();
    }

    // shallow copy
    int[] intsCopy = ints.clone();

    // deep copy
    HttpCookie[] cookiesCopy = new HttpCookie[cookies.length];

    for (int i = 0; i < cookies.length; i++) {
        // manually create copy of each element in array
        cookiesCopy[i] = cookies[i].clone();
    }

    doLogic(intsCopy, cookiesCopy);
}
```

Note that defensive copying applies to outputs as well. Return a copy of any mutable object stored in a private field from a method, unless the method explicitly specifies that it returns a direct reference to the object. Attackers given a direct reference to an internally stored mutable object can modify it after the method has returned.

Clearly, mutable (including non-final) inputs and outputs place a significant burden on method implementations. To minimize this burden, favor immutability when designing new classes [8]. In addition, if a class merely serves as a container for mutable inputs or outputs (the class does not directly operate on them), it may not be necessary to create defensive copies. For example, arrays and the standard collection classes do not create copies of caller-provided values. If a copy is desired so updates to a value do not affect the corresponding value in the collection, the caller must create the copy before inserting it into the collection, or after receiving it from the collection.

Guideline 2-2 Support copy functionality for a mutable class

When designing a mutable class, provide a means to create copies of its instances. This allows instances of that class to be safely passed to or returned from methods in other classes (see Guideline 2-1). This functionality may be provided by a copy constructor, or by implementing the java.lang.Cloneable interface and declaring a public `clone` method. A well-behaved `clone` implementation first calls `super.clone`. It then replaces, as necessary, any internal mutable object in the clone with a copy of that object. This ensures the returned clone is independent of the original object (changes to the clone will not affect the original, and vice versa).

If a class is final and does not provide an accessible method for acquiring a copy of it, callers can resort to performing a manual copy. This involves retrieving state from an instance of that class, and then creating a new instance with the retrieved state. Mutable state retrieved during this process must likewise be copied if necessary. Performing such a manual copy can be fragile. If the class evolves to include additional state in the future, then manual copies may not include that state.

Guideline 2-3 Validate inputs

Attacks using maliciously crafted inputs are well-documented [2] [6]. Such attacks often involve the manipulation of an input string format, the injection of information into a request parameter, or the overflow of an integer value. Validate inputs to prevent such malicious values from causing a vulnerability. Note that input validation must occur after any defensive copying of that input (see Guideline 2-1).

3 Classes

Guideline 3-1 Treat public static fields as constants

Callers can trivially access and modify public non-final static fields. Neither accesses nor modifications can be checked by a SecurityManager, and newly set values can not be validated. Treat a public static field as a constant. Declare it final, and only store an immutable value in the field:

```
public final class Directions {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
}
```

Constants can alternatively be defined using an *enum* declaration.

Guideline 3-2 Define wrapper methods around modifiable internal state

If state internal to a class must be publically accessible and modifiable, declare a private field and enable access to it via public wrapper methods (for static state, declare a private static field and public static wrapper methods). If the state is only intended to be accessed by subclasses, declare a private field and enable access via protected wrapper methods. Wrapper methods allow SecurityManager checks and input validation to occur prior to the setting of a new value:

```
public final class WrappedState {

    // private immutable object
    private String state;

    // wrapper method
    public String getState() {
        return state;
    }

    // wrapper method
    public void setState(String newState) {
        // permission needed to set state
        securityManagerCheck();
        inputValidation(newState);
        state = newState;
    }

}
```

Make additional defensive copies in `getState` and `setState` if the internal state is mutable:

```
public final class WrappedMutableState {

    // private mutable object
    private HttpCookie myState;

    // wrapper method
    public HttpCookie getState() {
        if (myState == null) {
            return null;
        } else {
            // copy
            return myState.clone();
        }
    }

    // wrapper method
    public void setState(HttpCookie newState) {

        // permission needed to set state
        securityManagerCheck();

        if (newState == null) {
            myState = null;
        } else {
            // copy
            newState = newState.clone();
            inputValidation(newState);
            myState = newState;
        }
    }

}
```

Guideline 3-3 Define wrappers around native methods

Java code is subject to oversight by the SecurityManager. Native code, on the other hand, is not. In addition, while pure Java code is immune to traditional buffer overflow attacks, native methods are not. To offer some of these protections during the invocation of native code, do not declare a native method public. Instead, declare it private and wrap it inside a public Java-based accessor method. A wrapper can enforce a preliminary SecurityManager check and perform any necessary input validation prior to the invocation of the native method:

```
public final class NativeMethodWrapper {

    // private native method
    private native void nativeOperation(byte[] data, int offset, int len);


    // wrapper method performs checks
    public void doOperation(byte[] data, int offset, int len) {
        // permission needed to invoke native method
        securityManagerCheck();

        if (data == null) {
            throw new NullPointerException();
        }

        // copy mutable input
        data = data.clone();

        // validate input
        if (offset < 0 || len < 0 || offset > data.length -□□ len) {
            throw new IllegalArgumentException();
        }

        nativeOperation(data, offset, len);
    }

}
```

Guideline 3-4 Purge sensitive information from exceptions

Exception objects can convey sensitive information. If a method calls the `java.io.FileInputStream` constructor to read an underlying configuration file and that file is not present, for example, a `FileNotFoundException` containing the file path is thrown. Propagating this exception back to the method caller exposes the layout of the file system. Exposing a file path containing the current user's name or home directory exacerbates the problem. `SecurityManager` checks guard this same information in standard system properties, and revealing it in exception messages effectively allows these checks to be bypassed.

Catch and sanitize internal exceptions before propagating them to upstream callers. Both the exception message and exception type can reveal sensitive information. A `FileNotFoundException` exposes a file system's layout in its message, and a specific file's absence via its type. Catch and throw a new instance of the same exception (with a sanitized message) when merely the message exposes sensitive information. Otherwise, throw a different type of exception and message altogether.

Do not sanitize exceptions containing information derived from caller inputs. If a caller provides the name of a file to be opened, for example, do not sanitize any resulting `FileNotFoundException` thrown when attempting to open that file.

4 Object Construction

Guideline 4-1 Prevent the unauthorized construction of sensitive classes

Limit the ability to construct instances of security-sensitive classes, such as `java.lang.ClassLoader`. A security-sensitive class enables callers to modify or circumvent SecurityManager access controls. Any instance of `ClassLoader`, for example, has the power to define classes with arbitrary security permissions.

To restrict untrusted code from instantiating a class, enforce a SecurityManager check at all points where that class can be instantiated. In particular, enforce a check at the beginning of each public and protected constructor. In classes that declare public static factory methods in place of constructors, enforce checks at the beginning of each factory method. Also enforce checks at points where an instance of a class can be created without the use of a constructor. Specifically, enforce a check inside the `readObject` or `readObjectNoData` method of a serializable class, and inside the `clone` method of a cloneable class.

If the security-sensitive class is non-final, this guideline not only blocks the direct instantiation of that class, it blocks malicious subclassing as well.

Guideline 4-2 Defend against partially initialized instances of non-final classes

If a constructor in a non-final class throws an exception, attackers can attempt to gain access to partially initialized instances of that class. Ensure a non-final class remains totally unusable until its constructor completes successfully.

One potential solution involves the use of an *initialized* flag. Set the flag as the last operation in a constructor before returning successfully. All overridable methods in the class must first consult the flag before proceeding:

```
// non-final java.lang.ClassLoader
public class ClassLoader {

    // initialized flag
    private volatile boolean initialized = false;

    protected ClassLoader() {
        // permission needed to create ClassLoader
        securityManagerCheck();
        init();

        // last step
        initialized = true;
    }

    protected final Class defineClass(...) {
        if (!initialized) {
            throw new SecurityException("object not initialized");
        }

        // regular logic follows
    }
}
```

Partially initialized instances of a non-final class can be accessed via a finalizer attack. The attacker overrides the protected `finalize` method in a subclass, and attempts to create a new instance of that subclass. This attempt fails (in the above example, the `SecurityManager` check in `ClassLoader`'s constructor throws a security exception), but the attacker simply ignores any exception and waits for the virtual machine to perform finalization on the partially initialized object. When that occurs the malicious `finalize` method implementation is invoked, giving the attacker access to this, a reference to the object being finalized. Although the object is only partially initialized, the attacker can still invoke methods on it (thereby circumventing the `SecurityManager` check). While the `initialized` flag does not prevent access to the partially initialized object, it does prevent methods on that object from doing anything useful for the attacker.

Use of an *initialized* flag, while secure, can be cumbersome. Simply ensuring that all fields in a public non-final class contain a safe value (such as null) until object initialization completes successfully can represent a reasonable alternative in classes that are not security-sensitive.

Guideline 4-3 Prevent constructors from calling methods that can be overridden

Do not call methods that can be overridden from a constructor, since that gives attackers a reference to `this` (the object being constructed) before the object has been fully initialized. Likewise, do not invoke methods that can be overridden from `clone`, `readObject`, or `readObjectNoData`. Otherwise attacks against partially initialized objects can be mounted in those cases as well.

5 Serialization and Deserialization

Guideline 5-1 Guard sensitive data during serialization

Once a class has been serialized the Java language's access controls can no longer be enforced (attackers can access private fields in an object by analyzing its serialized byte stream). Therefore do not serialize sensitive data in a serializable class.

Declare a sensitive field transient if relying on default serialization. Alternatively, implement the `writeObject`, `writeReplace`, or `writeExternal` method, and ensure the method implementation does not write sensitive fields to the serialized stream, or define the `serialPersistentFields` array field and ensure sensitive fields are not added to the array.

Guideline 5-2 View deserialization the same as object construction

Deserialization creates a new instance of a class without invoking any constructor on that class. Perform the same input validation checks in a *readObject* method implementation as those performed in a constructor. Likewise, assign default values consistent with those assigned in a constructor to all fields, including transient fields, not explicitly set during deserialization.

In addition, create copies of deserialized mutable objects before assigning them to internal fields in a `readObject` implementation. This defends against hostile code from deserializing byte streams that are specially crafted to give the attacker references to mutable objects inside the deserialized container object [8].

Attackers can also craft hostile streams in an attempt to exploit partially initialized (deserialized) objects. Ensure a serializable class remains totally unusable until deserialization completes successfully, for example by using an `initialized` flag. Declare the flag as a private transient field, and only set it in a `readObject` or `readObjectNoData` method (and in constructors) just prior to returning successfully. All public and protected methods in the class must consult the `initialized` flag before proceeding with their normal logic. As discussed earlier, use of an `initialized` flag can be cumbersome. Simply ensuring that all fields contain a safe value (such as null) until deserialization successfully completes can represent a reasonable alternative.

Guideline 5-3 Duplicate the SecurityManager checks enforced in a class during serialization and deserialization

Prevent an attacker from using serialization or deserialization to bypass the SecurityManager checks enforced in a class. Specifically, if a serializable class enforces a SecurityManager check in its constructors, then enforce that same check in a `readObject` or `readObjectNoData` method implementation. Otherwise an instance of the class can be created without any check via deserialization.

```
public final class SensitiveClass implements java.io.Serializable {

    public SensitiveClass() {
        // permission needed to instantiate SensitiveClass
        securityManagerCheck();

        // regular logic follows
    }

    // implement readObject to enforce checks during deserialization
    private void readObject(java.io.ObjectInputStream in) {
        // duplicate check from constructor
        securityManagerCheck();

        // regular logic follows
    }
}
```

If a serializable class enables internal state to be modified by a caller (via a public method, for example), and the modification is guarded with a SecurityManager check, then enforce that same check in a `readObject` method implementation. Otherwise, an attacker can use deserialization to create another instance of an object with modified state without passing the check.

```
public final class SecureName implements java.io.Serializable {

    // private internal state
    private String name;

    private static final String DEFAULT = "DEFAULT";

    public SecureName() {
        // initialize name to default value
        name = DEFAULT;
    }

    // allow callers to modify private internal state
    public void changeName(String newName) {
        if (name.equals(newName)) {
            // no change - do nothing
            return;
        } else {
            // permission needed to modify name
            securityManagerCheck();

            inputValidation(newName);

            name = newName;
        }
    }

    // implement readObject to enforce checks during deserialization
    private  readObject(java.io.ObjectInputStream in) {
        defaultReadObject();

        // if the deserialized name does not match the default value
normally
        // created at construction time, duplicate checks

        if (!DEFAULT.equals(name)) {
            securityManagerCheck();
            inputValidation(name);
        }
    }

}
```

If a serializable class enables internal state to be retrieved by a caller, and the retrieval is guarded with a SecurityManager check to prevent disclosure of sensitive data, then enforce that same check in a *writeObject* method implementation. Otherwise, an attacker can serialize an object to bypass the check and access the internal state (simply by reading the serialized byte stream).

```
public final class SecureValue implements java.io.Serializable {

    // sensitive internal state
    private String value;

    // public method to allow callers to retrieve internal state
    public String getValue() {
        // permission needed to get value
        securityManagerCheck();
        return value;
    }


    // implement writeObject to enforce checks during serialization
    private void writeObject(java.io.ObjectOutputStream out) {
        // duplicate check from getValue()
        securityManagerCheck();
        out.writeObject(value);
    }

}
```

6 Standard APIs

Guideline 6-1 Safely invoke java.security.AccessController.doPrivileged

`AccessController.doPrivileged` enables code to exercise its own permissions when performing SecurityManager-checked operations. To avoid inadvertently performing such operations on behalf of unauthorized callers, do not invoke `doPrivileged` using caller-provided inputs (tainted inputs):

```
import java.io.*;
import java.security.*;

    private static final String FILE = "/myfile";

    public FileInputStream getFile() {
        return (FileInputStream)AccessController.doPrivileged(new
PrivilegedAction() {
            public Object run() {
                return new FileInputStream(FILE); // checked by
SecurityManager
            }
        });
    }
```

The implementation of `getFile` properly opens the file using a hardcoded value. More specifically, it does not allow the caller to influence the name of the file to be opened by passing a caller-provided (tainted) input to `doPrivileged`.

Caller inputs that have been validated can sometimes be safely used with `doPrivileged`. Typically the inputs must be restricted to a limited set of acceptable (usually hardcoded) values.

Guideline 6-2 Safely invoke standard APIs that bypass SecurityManager checks depending on the immediate caller's class loader

Certain standard APIs in the core libraries of the Java runtime enforce SecurityManager checks, but allow those checks to be bypassed depending on the immediate caller's class loader. When the `java.lang.Class.newInstance` method is invoked on a Class object, for example, the immediate caller's class loader is compared to the Class object's class loader. If the caller's class loader is an ancestor of (or the same as) the Class object's class loader, the `newInstance` method bypasses a SecurityManager check (see Section 4.3.2 in [7] for information on class loader relationships). Otherwise, the relevant SecurityManager check is enforced.

The difference between this class loader comparison and a SecurityManager check is noteworthy. A SecurityManager check investigates all callers in the current execution chain to ensure each has been granted the requisite security permission (if `AccessController.doPrivileged` was invoked in the chain, all callers leading back to the caller of `doPrivileged` are checked). In contrast, the class loader comparison only investigates the immediate caller's context (its class loader). This means any caller who invokes `Class.newInstance` and who has the capability to pass the class loader check - thereby bypassing the SecurityManager - effectively performs the invocation inside an implicit `AccessController.doPrivileged` action. Because of this subtlety, callers should ensure they do not inadvertently invoke `Class.newInstance`on behalf of untrusted code.

The following methods behave similar to `Class.newInstance`, and potentially bypass SecurityManager checks depending on the immediate caller's class loader:

```
java.lang.Class.newInstance
java.lang.Class.getClassLoader
java.lang.Class.getClasses
java.lang.Class.getField(s)
java.lang.Class.getMethod(s)
java.lang.Class.getConstructor(s)
java.lang.Class.getDeclaredClasses
java.lang.Class.getDeclaredField(s)
java.lang.Class.getDeclaredMethod(s)
java.lang.Class.getDeclaredConstructor(s)
java.lang.ClassLoader.getParent
java.lang.ClassLoader.getSystemClassLoader
java.lang.Thread.getContextClassLoader
```

Refrain from invoking the above methods on Class, ClassLoader, or Thread instances received from untrusted code. If the respective instances were acquired safely (or in the case of the static `ClassLoader.getSystemClassLoader` method), do not invoke the above methods using inputs provided by untrusted code. Also, do not propagate objects returned by the above methods back to untrusted code.

Guideline 6-3 Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance

The following static methods perform tasks using the immediate caller's class loader:

```
java.lang.Class.forName
java.lang.Package.getPackage(s)
java.lang.Runtime.load
java.lang.Runtime.loadLibrary
java.lang.System.load
java.lang.System.loadLibrary
java.sql.DriverManager.getConnection
java.sql.DriverManager.getDriver(s)
java.sql.DriverManager.deregisterDriver
java.util.ResourceBundle.getBundle
```

For example, `System.loadLibrary("/com/foo/MyLib.so")` uses the immediate caller's class loader to find and load the specified library (loading libraries enables a caller to make native method invocations). Do not invoke this method on behalf of untrusted code, since untrusted code may not have the ability to load the same library using its own class loader instance. Do not invoke any of these methods using inputs provided by untrusted code, and do not propagate objects returned by these methods back to untrusted code.

Guideline 6-4 Be aware of standard APIs that perform Java language access checks against the immediate caller

When an object access fields or methods in another object, the virtual machine automatically performs language access checks (it prevents objects from invoking private methods in other objects, for example).

Code may also call standard APIs (primarily in the java.lang.reflect package) to reflectively access fields or methods in another object. The following reflection-based APIs mirror the language checks enforced by the virtual machine:

```
java.lang.Class.newInstance
java.lang.reflect.Constructor.newInstance
java.lang.reflect.Field.get*
java.lang.reflect.Field.set*
java.lang.reflect.Method.invoke
java.util.concurrent.atomic.AtomicIntegerFieldUpdater.newUpdater
java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater
java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater
```

Language checks are performed solely against the immediate caller (not against each caller in the execution sequence). Since the immediate caller may have capabilities that other code lacks (it may belong to a particular package and therefore have access to its package-private members), do not invoke the above APIs on behalf of untrusted code. Specifically, do not invoke the above methods on Class, Constructor, Field, or Method instances received from untrusted code. If the respective instances were acquired safely, do not invoke the above methods using inputs provided by untrusted code. Also, do not propagate objects returned by the above methods back to untrusted code.

References

1. William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin.
   Firewalls andInternet Security.
   2nd ed. Boston, MA: Addison-Wesley, 2003.
2. Gary McGraw. Software Security: Building Security In.
   Boston: Addison-Wesley, 2006.
3. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.
   The Java Language Specification.
   3rd ed. Boston, MA: Addison-Wesley, 2005.
4. Tim Lindholm and Frank Yellin.
   The Java Virtual Machine Specification. 2nd ed.
   Reading, MA: Addison-Wesley, 1999.
5. Aleph One. Smashing the Stack for Fun and Profit.
   Phrack 49, November 1996.
6. John Viega and Gary McGraw.
   Building Secure Software: How to Avoid Security Problems the Right Way.
   Boston: Addison-Wesley, 2002.
7. Li Gong, Gary Ellison, and Mary Dageforde.
   Inside Java 2 Platform Security. 2nd ed.
   Boston, MA: Addison-Wesley, 2003.
8. Joshua Bloch. Effective Java Programming Language Guide.
   1st ed. Addison-Wesley Professional, 2001.