Secure Coding Guidelines for the Java Programming Language, Version 4.0

-
-
-
-
-
-
-
-
-
-
-
-
-

Introduction

One of the main design considerations for the Java platform is to provide a secure environment for executing mobile code. Java comes with its own unique set of security challenges. While the Java security architecture [1] can protect users and systems from hostile programs downloaded over a network, it cannot defend against implementation bugs that occur in *trusted* code. Such bugs can inadvertently open the very holes that the security architecture was designed to contain, including access to files, printers, webcams, microphones, and the network from behind firewalls. In severe cases local programs may be executed or Java security disabled. These bugs can potentially be used to turn the machine into a zombie computer, steal confidential data from machine and intranet, spy through attached devices, prevent useful operation of the machine, assist further attacks, and many other malicious activities.

The choice of language system impacts the robustness of any software program. The Java language [2] and virtual machine [3] provide many features to mitigate common programming mistakes. The language is type-safe, and the runtime provides automatic memory management and bounds-checking on arrays. Java programs and libraries check for illegal state at the earliest opportunity. These features also make Java programs immune to the stack-smashing [4] and buffer overflow attacks possible in the C and to a lesser extent C++ programming languages. These attacks have been described as the single most pernicious problem in computer security today [5].

The explicit static typing of Java makes code easy to understand and the dynamic checks ensure unexpected conditions result in predictable behaviour -- which makes Java a joy to use.

To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should adhere to recommended coding guidelines. Existing publications, such as Effective Java [6], provide excellent guidelines related to Java software design. Others, such as Software Security: Building Security In [7], outline guiding principles for software security. This paper bridges such publications together and includes coverage of additional topics. This document provides a more complete set of security-specific coding guidelines targeted at the Java programming language. These guidelines are of interest to all Java developers, whether they create trusted end-user applications and applets, implement the internals of a security component, or develop shared Java class libraries that perform common programming tasks. Any implementation bug can have serious security ramifications and could appear in any layer of the software stack.

0 Fundamentals

The following general principles apply throughout Java security.

Guideline 0-0: Prefer to have obviously no flaws rather than no obvious flaws [8]

Creating secure code is not necessarily easy. Despite the unusually robust nature of Java, flaws can slip past with surprising ease. Design and write code that does not require clever logic to see that it is safe. Specifically, follow the guidelines in this document unless there is a very strong reason not to.

Guideline 0-1: Design APIs to avoid security concerns

It is better to design APIs with security in mind. Trying to retrofit security into an existing API is more difficult and error prone. For example, making a class final prevents a malicious subclass from adding finalizers, cloning, and overriding random methods (Guideline 4-5). Any use of the `SecurityManager` highlights an area that should be scrutinized.

Guideline 0-2: Avoid duplication

Duplication of code and data causes many problems. Both code and data tend not to be treated consistently when duplicated, e.g., changes may not be applied to all copies.

Guideline 0-3: Restrict privileges

Despite best efforts, not all coding flaws will be eliminated even in well reviewed code. However, if the code is operating with reduced privileges, then exploitation of any flaws is likely to be thwarted. The most extreme form of this is known as the principle of least privilege. Using the Java security mechanism this can be implemented statically by restricting permissions through policy files and dynamically with the use of the `java.security.AccessController.doPrivileged` mechanism (see section 6). For applets and JNLP applications the best approach is often to leave the jar files unsigned. The application then runs in a sandbox and will not be able to execute any potentially dangerous code. The application will be safe for the user to run because if it attempts to execute security-sensitive code, the JRE will throw a security exception.

Guideline 0-4: Establish trust boundaries

In order to ensure that a system is protected, it is necessary to establish trust boundaries. Data that crosses these boundaries should be sanitized and validated before use. Trust boundaries are also necessary to allow security auditing to be performed efficiently. Code that ensures integrity of trust boundaries must itself be loaded in such a way that its own integrity is assured.

For instance, a web browser is outside of the system for a web server. Equally, a web server is outside of the system for a web browser. Therefore, web browser and server software should not rely upon the behavior of the other for security.

When auditing trust boundaries, there are some questions that should be kept in mind. Are the code and data used sufficiently trusted? Could a library be replaced with a malicious implementation? Is untrusted configuration data being used? Is code calling with lower privileges adequately protected against?

Guideline 0-5: Minimise the number of permission checks

Java is primarily an object-capability language. `SecurityManager` checks should be considered a last resort. Perform security checks at a few defined points and return an object (a capability) that client code retains so that no further permission checks are required.

Guideline 0-6: Encapsulate

Allocate behaviors and provide succinct interfaces. Fields of objects should be private and accessors avoided. The interface of a method, class, package, and module should form a coherent set of behaviors, and no more.

1 Denial of Service

Input into a system should be checked so that it will not cause excessive resource consumption disproportionate to that used to request the service. Common affected resources are CPU cycles, memory, disk space, and file descriptors.

In rare cases it may not be practical to ensure that the input is reasonable. It may be necessary to carefully combine the resource checking with the logic of processing the data. For client systems it is generally left to the user to close the application if it is using excessive resources. Therefore, only attacks resulting in persistent DoS, such as wasting significant disk space, need be defended against. Server systems should be robust against external attacks.

Guideline 1-1: Beware of activities that may use disproportionate resources

Examples of attacks include:

- Requesting a large image size for vector graphics. For instance, SVG and font files.
- Integer overflow errors can cause sanity checking of sizes to fail.
- An object graph constructed by parsing a text or binary stream may have memory requirements many times that of the original data.
- "Zip bombs" whereby a short file is very highly compressed. For instance, ZIPs, GIFs and gzip encoded http contents. When decompressing files it is better to set limits on the decompressed data size rather than relying upon compressed size or meta-data.
- "Billion laughs attack" whereby XML entity expansion causes an XML document to grow dramatically during parsing. Set the `XMLConstants.FEATURE_SECURE_PROCESSING` feature to enforce reasonable limits.
- Causing many keys to be inserted into a hash table with the same hash code, turning an algorithm of around O(n) into O(n^2).
- Regular expressions may exhibit catastrophic backtracking.
- XPath expressions may consume arbitrary amounts of processor time.
- Java deserialization and Java Beans XML deserialization of malicious data may

result in unbounded memory or CPU usage.

- Detailed logging of unusual behavior may result in excessive output to log files.
- Infinite loops can be caused by parsing some corner case data. Ensure that each iteration of a loop makes some progress.

Guideline 1-2: Release resources in all cases

Exceptions are often overlooked. Resources should always be released promptly even if an exception is thrown. Java SE 7 has a syntax enhancement (try-with-resources) for automatically handling the release of many resources.

```
        try (final InputStream in = new FileInputStream(file))
{
            use(in);
        }
```

For Java SE 6 code and resources without support for the enhanced feature, use the standard resource acquisition and release. Attempts to rearrange this idiom typically result in errors and makes the code significantly harder to follow.

```
        final InputStream in = new FileInputStream(file);
        try {
            use(in);
        } finally {
            in.close();
        }
```

Ensure that any output buffers are flushed in the case that output was otherwise successful. If the flush fails, the code should exit via an exception.

```
        final OutputStream rawOut = new FileOutputStream(file);
        try {
            final BufferedOutputStream out =
                new BufferedOutputStream(rawOut);
            use(out);
            out.flush();
        } finally {
            rawOut.close();
        }
```

Guideline 1-3: Resource limit checks should not suffer from integer overflow

The Java language provides bounds checking on arrays which mitigates the vast majority of integer overflow attacks. However, all the primitive integer types silently overflow. Therefore, take care when checking resource limits. This is particularly important on persistent resources, such as disk space, where a reboot may not clear the problem.

Some checking can be rearranged so as to avoid overflow. With large values, `current + max` could overflow to a negative value, which would always be less than `max`.

```
        private void checkGrowBy(long extra) {
            if (extra < 0 || current > max - extra) {
                  throw new IllegalArgumentException();
            }
        }
```

If performance is not a particular issue, a verbose approach is to use arbitrary sized integers.

```
        private void checkGrowBy(long extra) {
            BigInteger currentBig = BigInteger.valueOf(cur-
rent);
            BigInteger maxBig     = BigInteger.valueOf(max
);
            BigInteger extraBig   = BigInteger.valueOf(extra
);

            if (extra < 0 ||
                currentBig.add(extraBig).compareTo(maxBig) > 0)
{
                  throw new IllegalArgumentException();
            }
        }
```

A peculiarity of two's complement integer arithmetic is that the minimum negative value does not have a matching positive value of the same magnitude. So, `Integer.MIN_VALUE == -Integer.MIN_VALUE`, `Integer.MIN_VALUE ==`

`Math.abs(Integer.MIN_VALUE)` and, for integer `a`, `a < 0` does not imply `-a > 0`. The same edge case occurs for `Long.MIN_VALUE`.

2 Confidential Information

Confidential data should be readable only within a limited context. Data that is to be trusted should not be exposed to tampering. Privileged code should not be executable through intended interfaces.

Guideline 2-1: Purge sensitive information from exceptions

Exception objects may convey sensitive information. For example, if a method calls the `java.io.FileInputStream` constructor to read an underlying configuration file and that file is not present, a `java.io.FileNotFoundException` containing the file path is thrown. Propagating this exception back to the method caller exposes the layout of the file system. Many forms of attack require knowing or guessing locations of files.

Exposing a file path containing the current user's name or home directory exacerbates the problem. `SecurityManager` checks guard this information when it is included in standard system properties (such as `user.home`) and revealing it in exception messages effectively allows these checks to be bypassed.

Internal exceptions should be caught and sanitized before propagating them to upstream callers. The type of an exception may reveal sensitive information, even if the message has been removed. For instance, `FileNotFoundException` reveals whether or not a given file exists.

It is not necessary to sanitize exceptions containing information derived from caller inputs. If a caller provides the name of a file to be opened, there is no need to sanitize any resulting `FileNotFoundException` thrown. In this case the exception provides useful debugging information.

Be careful when depending on an exception for security because its contents may change in the future. Suppose a previous version of a library did not include a potentially sensitive piece of information in the exception, and an existing client relied upon that for security. For example, a library may throw an exception without a message. An application programmer may look at this behavior and decide that it is okay to propagate the exception. However, a later version of the library may add extra debugging information to the exception message. The application exposes this additional information, even though the application code itself may not have changed. Only include known, acceptable information from an exception rather than filtering out some elements of the exception.

Exceptions may also include sensitive information about the configuration and internals of the system. Do not pass exception information to end users unless one knows exactly what it contains. For example, do not include exception stack traces inside HTML comments.

Guideline 2-2: Do not log highly sensitive information

Some information, such as Social Security numbers (SSNs) and passwords, is highly sensitive. This information should not be kept for longer than necessary nor where it may be seen, even by administrators. For instance, it should not be sent to log files and its presence should not be detectable through searches. Some transient data may be kept in mutable data structures, such as char arrays, and cleared immediately after use. Clearing data structures has reduced effectiveness on typical Java runtime systems as objects are moved in memory transparently to the programmer.

This guideline also has implications for implementation and use of lower-level libraries that do not have semantic knowledge of the data they are dealing with. As an example, a low-level string parsing library may log the text it works on. An application may parse an SSN with the library. This creates a situation where the SSNs are available to administrators with access to the log files.

Guideline 2-3: Consider purging highly sensitive from memory after use

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

3 Injection and Inclusion

A very common form of attack involves causing a particular program to interpret data crafted in such a way as to cause an unanticipated change of control. Typically, but not always, this involves text formats.

Guideline 3-1: Generate valid formatting

Attacks using maliciously crafted inputs to cause incorrect formatting of outputs are well-documented [7]. Such attacks generally involve exploiting special characters in an input string, incorrect escaping, or partial removal of special characters.

If the input string has a particular format, combining correction and validation is highly error-prone. Parsing and canonicalization should be done before validation. If possible, reject invalid data and any subsequent data, without attempting correction. For instance, many network protocols are vulnerable to cross-site POST attacks, by interpreting the HTTP body even though the HTTP header causes errors.

Use well-tested libraries instead of ad hoc code. There are many libraries for creating XML. Creating XML documents using raw text is error-prone. For unusual formats where appropriate libraries do not exist, such as configuration files, create classes that cleanly handle all formatting and only formatting code.

Guideline 3-2: Avoid dynamic SQL

It is well known that dynamically created SQL statements including untrusted input are subject to command injection. This often takes the form of supplying an input containing a quote character (`'`) followed by SQL. Avoid dynamic SQL.

For parameterised SQL statements using Java Database Connectivity (JDBC), use `java.sql.PreparedStatement` or `java.sql.CallableStatement` instead of `java.sql.Statement`. In general, it is better to use a well-written, higher-level library to insulate application code from SQL. When using such a library, it is not necessary to limit characters such as quote (`'`). If text destined for XML/HTML is handled correctly during output (Guideline 3-3), then it is unnecessary to disallow characters such as less than (<) in inputs to SQL.

Guideline 3-3: XML and HTML generation requires care

Cross Site Scripting (XSS) is a common vulnerability in web applications. This is generally caused by inputs included in outputs without first validating the input. For example, checking for illegal characters and escaping data properly. It is better to use a library that constructs XML or HTML rather than attempting to insert escape codes in every field in every document. In particular, be careful when using Java Server Pages (JSP).

Guideline 3-4: Avoid any untrusted data on the command line

When creating new processes, do not place any untrusted data on the command line. Behavior is platform-specific, poorly documented, and frequently surprising. Malicious data may, for instance, cause a single argument to be interpreted as an option (typically a leading – on Unix or / on Windows) or as two separate arguments. Any data that needs to be passed to the new process should be passed either as encoded arguments (e.g., Base64), in a temporary file, or through a inherited channel.

Guideline 3-5: Restrict XML inclusion

XML Document Type Definitions (DTDs) allow URLs to be defined as system entities, such as local files and http URLs within the local intranet or localhost. XML External Entity (XXE) attacks insert local files into XML data which may then be accessible to the client. Similar attacks may be made using XInclude, the XSLT document function, and the XSLT import and include elements. The safe way to avoid these problems whilst maintaining the power of XML is to reduce privileges as described in Guideline 9-2. You may decide to give some access through this technique, such as inclusion to pages from the same-origin web site. Another approach, if such an API is available, is to set all entity resolvers to safe implementations.

Note that this issue generally applies to the use of APIs that use XML but are not specifically XML APIs.

Guideline 3-6: Care with BMP files

BMP images files may contain references to local ICC files. Whilst the contents of ICC files is unlikely to be interesting, the act of attempting to read files may be an issue. Either avoid BMP files, or reduce privileges as Guideline 9-2.

Guideline 3-7: Disable HTML display in Swing components

Many Swing pluggable look-and-feels interpret text in certain components starting with `<html>` as HTML. If the text is from an untrusted source, an adversary may craft the HTML such that other components appear to be present or to perform inclusion attacks.

To disable the HTML render feature, set the `"html.disable"` client property of each component to `Boolean.TRUE` (no other `Boolean true` instance will do).

```
        label.putClientProperty("html.disable", true);
```

Guideline 3-8: Take care interpreting untrusted code

Code can be hidden in a number of places. If the source is untrusted then a secure sandbox must be constructed to run it in. Some examples of components or APIs that can potentially execute untrusted code include:

- Scripts run through the scripting API or similar.
- By default the Sun implementation of the XSLT interpreter enables extensions to call Java code. Set the `javax.xml.XMLConstants.FEATURE_SECURE_PROCESSING` feature to disable it.
- Long Term Persistence of JavaBeans Components supports execution of Java statements.
- RMI and LDAP (RFC 2713) may allow loading of remote code specified by remote connection. On the Oracle JDK, RMI remote code loading may be disabled by setting the `"java.rmi.server.codebase"` system property. LDAP code loading is disabled unless the `"com.sun.jndi.ldap.object.trustURLCodebase"` is set to `"true"`.
- Many SQL implementations allow execution of code with effects outside of the database itself.

4 Accessibility and Extensibility

The task of securing a system is made easier by reducing the "attack surface" of the code.

Guideline 4-1: 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 API, otherwise, declare it package-private. Similarly, declare class members and constructors (nested classes, methods, or fields) public or protected as appropriate, if they are also part of the API. Otherwise, declare them private or package-private to avoid exposing the implementation. Note that members of interfaces are implicitly public.

Classes loaded by different loaders do not have package-private access to one another even if they have the same package name. Classes in the same package loaded by the same class loader must either share the same code signing certificate or not have a certificate at all. In the Java virtual machine class loaders are responsible for defining packages. It is recommended that, as a matter of course, packages are marked as sealed in the jar file manifest.

Guideline 4-2: Limit the accessibility of packages

Containers may hide implementation code by adding to the `package.access` security property. This property prevents untrusted classes from other class loaders linking and using reflection on the specified package hierarchy. Care must be taken to ensure that packages cannot be accessed by untrusted contexts before this property has been set.

This example code demonstrates how to append to the `package.access` security property. Note that it is not thread-safe. This code should generally only appear once in a system.

```
        private static final String PACKAGE_ACCESS_KEY = "pack-
age.access";
        static {
            String packageAccess = java.security.Security.get-
Property(
                PACKAGE_ACCESS_KEY
            );
            java.security.Security.setProperty(
                PACKAGE_ACCESS_KEY,
                (
                    (packageAccess == null ||
                     packageAccess.trim().isEmpty()) ?
                    "" :
                    (packageAccess + ",")
                ) +
                "xx.example.product.implementation."
            );
        }
```

Guideline 4-3: Isolate unrelated code

Containers should isolate unrelated application code and prevent package-private access between code with different permissions. Even otherwise untrusted code is typically given permissions to access its origin, and therefore untrusted code from different origins should be isolated. The Java Plugin, for example, loads unrelated applets into separate class loader instances and runs them in separate thread groups.

Some apparently global objects are actually local to applet or application contexts. Applets loaded from different web sites will have different values returned from, for example, `java.awt.Frame.getFrames`. Such static methods (and methods on true globals) use information from the current thread and the class loaders of code on the stack to determine which is the current context. This prevents malicious applets from interfering with applets from other sites.

Mutable statics (see Guideline 6-11) and exceptions are common ways that isolation is inadvertently breached.

Guideline 4-4: Limit exposure of ClassLoader instances

Access to `ClassLoader` instances allows certain operations that may be undesirable:

- Access to classes that client code would not normally be able to access.
- Retrieve information in the URLs of resources (actually opening the URL is limited with the usual restrictions).
- Assertion status may be turned on and off.
- The instance may be casted to a subclass. `ClassLoader` subclasses frequently have undesirable methods.

Guideline 9-8 explains access checks made on acquiring `ClassLoader` instances through various Java library methods. Care should be taken when exposing a class loader through the thread context class loader.

Guideline 4-5: Limit the extensibility of classes and methods

Design classes and methods for inheritance or declare them final [6]. Left non-final, a class or method can be maliciously overridden by an attacker. A class that does not permit subclassing is easier to implement and verify that it is secure. Prefer composition to inheritance.

```
// Unsubclassable class with composed behavior.
        public final class SensitiveClass {
```

```
        private final Behavior behavior;

        // Hide constructor.
        private SensitiveClass(Behavior behavior) {
            this.behavior = behavior;
        }

        // Guarded construction.
        public static SensitiveClass newSensitiveClass(
            Behavior behavior
        ) {
            // ... validate any arguments ...

            // ... perform security checks ...

            return new SensitiveClass(behavior);
        }
    }
```

Malicious subclasses that override the `Object.finalize` method can resurrect objects even if an exception was thrown from the constructor. Low-level classes with constructors explicitly throwing a `java.security.SecurityException` are likely to have security issues. From JDK6 on, an exception thrown before the `java.lang.Object` constructor exits which prevents the finalizer from being called. Therefore, if subclassing is allowed and security manager permission is required to construct an object, perform the check before calling the super constructor. This can be done by inserting a method call as an argument to an alternative ("this") constructor invocation.

```
    public class NonFinal {

        // sole accessible constructor
        public NonFinal() {
            this(securityManagerCheck());
        }

        private NonFinal(Void ignored) {
            // ...
            }


            private static Void securityManagerCheck() {
                SecurityManager sm = System.getSecurityManag-
er();
                if (sm != null) {
                    sm.checkPermission(...);
                }
                return null;
            }

        }
```

For compatibility with versions of Java prior to JDK 6, check that the class has been initialized before every sensitive operation and before trusting any other instance of the class. It may be possible to see a partially initialized instance, so any variable should have a safe interpretation for the default value. For mutable classes, it is advisable to make an "initialized" flag volatile to create a suitable *happens-before* relationship.

```
        public class NonFinal {

            private volatile boolean initialized;

            // sole constructor
            public NonFinal() {
                securityManagerCheck();

                // ... initialize class ...

                // Last action of constructor.
                this.initialized = true;
            }

            public void doSomething() {
                checkInitialized();
            }
            private void checkInitialized() {
                if (!initialized) {
                    throw new SecurityException(
                        "NonFinal not initialized"
                    );
                }
            }
        }
```

When confirming 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`), because instances are scoped both by their class name as well as the class loader that defined the class.

Guideline 4-6: 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 that is 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) // Secu-
rityManager
                                         remove(key)   //
checks for these
```

```
                                                            // meth-
ods
```

The class `java.security.Provider` extends from `java.util.Properties`, and `Properties` extends from `java.util.Hashtable`. In this hierarchy, the `Provider` class 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) is 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.

Malicious subclasses may implement `java.lang.Cloneable`. Implementing this interface affects the behaviour of the subclass. A clone of a victim object may be made. The clone will be a shallow copy. The intrinsic lock and fields of the two objects will be different, but referenced objects will be the same. This allows an adversary to confuse the state of instances of the attacked class.

5 Input Validation

A feature of the culture of Java is that rigorous method parameter checking is used to improve robustness. More generally, validating external inputs is an important part of security.

Guideline 5-1: Validate inputs

Input from untrusted sources must be validated before use. Maliciously crafted inputs may cause problems, whether coming through method arguments or external streams. Examples include overflow of integer values and directory traversal attacks by including `"../"` sequences in filenames. Ease-of-use features should be separated from programmatic interfaces. Note that input validation must occur after any defensive copying of that input (see Guideline 6-2).

Guideline 5-2: Validate output from untrusted objects as input

In general method arguments should be validated but not return values. However, in the case of an upcall (invoking a method of higher level code) the returned value should be validated. Likewise, an object only reachable as an implementation of an upcall need not validate its inputs.

Guideline 5-3: Define wrappers around native methods

Java code is subject to runtime checks for type, array bounds, and library usage. Native code, on the other hand, is generally not. While pure Java code is effectively 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 expose the functionality through a public Java-based wrapper method. A wrapper can safely 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) {
                // copy mutable input
                data = data.clone();
                // validate input
                // Note offset+len would be subject to integer
overflow.
                // For instance if offset = 1 and len = Inte-
ger.MAX_VALUE,
                //   then offset+len == Integer.MIN_VALUE which
is lower
                //   than data.length.
                // Further,
                //   loops of the form
                //       for (int i=offset; i<offset+len; ++i)
{ ... }
                //   would not throw an exception or cause na-
tive code to
                //   crash.

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

                nativeOperation(data, offset, len);
            }
        }
```

6 Mutability

Mutability, whilst appearing innocuous, can cause a surprising variety of security problems.

Guideline 6-1: Prefer immutability for value types

Making classes immutable prevents the issues associated with mutable objects (described in subsequent guidelines) from arising in client code. Immutable classes should not be subclassable. Further, hiding constructors allows more flexibility in instance creation and caching. This means making the constructor private or default access ("package-private"), or being in a package controlled by the `package.access` security property. Immutable classes themselves should declare fields final and protect against any mutable inputs and outputs as described in Guideline 6-2. Construction of immutable objects can be made easier by providing builders (cf. Effective Java [6]).

Guideline 6-2: Create copies of mutable output values

If a method returns a reference to an internal mutable object, then client code may modify the internal state of the instance. Unless the intention is to share state, copy mutable objects and return the copy.

To create a copy of a trusted mutable object, call a copy constructor or the clone method:

```
public class CopyOutput {
    private final java.util.Date date;
    ...
    public java.util.Date getDate() {
        return (java.util.Date)date.clone();
    }
}
```

Guideline 6-3: Create safe copies of mutable and subclassable input values

Mutable objects may be changed after and even during the execution of a method or constructor call. Types that can be subclassed may behave incorrectly, inconsistently, and/or maliciously. If a method is not specified to operate directly on a mutable input parameter, create a copy of that input and perform the method logic on the copy. 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) [7] can be exploited where a mutable input contains one value during a SecurityManager check but a different value when the input is used later.

To create a copy of an untrusted mutable object, call a copy constructor or creation method:

```
public final class CopyMutableInput {
    private final Date date;

    // java.util.Date is mutable
    public CopyMutableInput(Date date) {
```

```
                // create copy
                this.date = new Date(date.getTime());
            }
        }
```

In rare cases it may be safe to call a copy method on the instance itself. For instance, `java.net.HttpCookie` is mutable but final and provides a public *clone* method for acquiring copies of its instances.

```
        public final class CopyCookie {

            // java.net.HttpCookie is mutable
            public void copyMutableInput(HttpCookie cookie) {
                // create copy
                cookie = (HttpCookie)cookie.clone(); // Http-
Cookie is final

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

It is safe to call `HttpCookie.clone` because it cannot be overridden with a malicious implementation. Date also provides a public clone method, but because the method is overrideable it can be trusted only if the Date object is from a trusted source. Some classes, such as `java.io.File`, are subclassable even though they appear to be immutable.

This guideline does not apply to classes that are designed to wrap a target object. For instance, `java.util.Arrays.asList` operates directly on the supplied array without copying.

In some cases, notably collections, a method may require a deeper copy of an input object than the one returned via that input's copy constructor or `clone` method. Instantiating an `ArrayList` with a collection, for example, produces a shallow copy of the original collection instance. Both the copy and the original share references to the same elements. If the elements are mutable, then a deep copy over the elements is required:

```
          // String is immutable.
          public void shallowCopy(Collection<String> strs) {
              strs = new ArrayList<String>(strs);
              doLogic(strs);
          }
          // Date is mutable.
          public void deepCopy(Collection<Date> dates) {
              Collection<Date> datesCopy = new ArrayList<Date>
(dates.size());
              for (Date date : dates) {
                  datesCopy.add(new java.util.Date(date.get-
Time()));
              }
              doLogic(datesCopy);
          }
```

Constructors should complete the deep copy before assigning values to a field. An object should never be in a state where it references untrusted data, even briefly. Further, objects assigned to fields should never have referenced untrusted data due to the dangers of unsafe publication.

Guideline 6-4: Support copy functionality for a mutable class

When designing a mutable value class, provide a means to create safe copies of its instances. This allows instances of that class to be safely passed to or returned from methods in other classes (see Guideline 6-2 and Guideline 6-3). This functionality may be provided by a static creation method, a copy constructor, or by implementing a public copy method (for final classes).

If a class is final and does not provide an accessible method for acquiring a copy of it, callers could 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, then manual copies may not include that state.

The `java.lang.Cloneable` mechanism is problematic and should not be used. Implementing classes must explicitly copy all mutable fields which is highly error-prone. Copied fields may not be final. The clone object may become available before field copying has completed, possibly at some intermediate stage. In non-final classes `Object.clone` will make a new instance of the potentially malicious subclass. Implementing `Cloneable` is an implementation detail, but appears in the public interface of the class.

Guideline 6-5: Do not trust identity equality when overridable on input reference objects

Overridable methods may not behave as expected.

For instance, when expecting identity equality behavior, `Object.equals` may be overridden to return true for different objects. In particular when used as a key in a map, an object may be able to pass itself off as a different object that it should not have access to.

If possible, use a collection implementation that enforces identity equality, such as `IdentityHashMap`.

```
        private final Map<Window,Extra> extras = new Identity-
HashMap<>();

        public void op(Window window) {
            // Window.equals may be overridden,
            // but safe as we are using IdentityHashMap
            Extra extra = extras.get(window);
        }
```

If such a collection is not available, use a package private key which an adversary does not have access to.

```
        public class Window {
            /* pp */ class PrivateKey {
                // Optionally, refer to real object.
                /* pp */ Window getWindow() {
                    return Window.this;
                }
            }
            /* pp */ final PrivateKey privateKey = new Pri-
vateKey();
```

```
            private final Map<Window.PrivateKey,Extra> extras =
                                                new WeakHashMap<>
();
            ...
        }

        public class WindowOps {
            public void op(Window window) {
                // Window.equals may be overridden,
                // but safe as we don't use it.
                Extra extra = extras.get(window.privateKey);
                ...
            }
        }
```

Guideline 6-6: Treat passing input to untrusted object as output

The above guidelines on output objects apply when passed to untrusted objects. Appropriate copying should be applied.

```
        private final byte[] data;

        public void writeTo(OutputStream out) throws IOExcep-
tion {
            // Copy (clone) private mutable data before send-
ing.
            out.write(data.clone());
        }
```

A common but difficult to spot case occurs when an input object is used as a key. A collection's use of equality may well expose other elements to a malicious input object on or after insertion.

Guideline 6-7: Treat output from untrusted object as input

The above guidelines on input objects apply when returned from untrusted objects. Appropriate copying and validation should be applied.

```
        private final Date start;
        private Date end;

        public void endWith(Event event) throws IOException {
            Date end = new Date(event.getDate().getTime());
            if (end.before(start)) {
                throw new IllegalArgumentException("...");
            }
            this.end = end;
        }
```

Guideline 6-8: Define wrapper methods around modifiable internal state

If a state that is internal to a class must be publicly accessible and modifiable, declare a private field and enable access to it via public 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 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(final String newState) {
                this.state = requireValidation(newState);
            }

            private static String requireValidation(final
String state) {
                if (...) {
                    throw new IllegalArgumentException("...");
                }
                return state;
            }
        ,
    }
```

Make additional defensive copies in `getState` and `setState` if the internal state is mutable, as described in Guideline 6-2.

Where possible make methods for operations that make sense in the context of the interface of the class rather than merely exposing internal implementation.

Guideline 6-9: Make public static fields final

Callers can trivially access and modify public non-final static fields. Neither accesses nor modifications can be guarded against, and newly set values cannot be validated. Fields with subclassable types may be set to objects with malicious implementations. Always declare public static fields as final.

```
    public class Files {
        public static final String separator = "/";
        public static final String pathSeparator = ":";
    }
```

If using an interface instead of a class, the "`public static final`" can be omitted to improve readability, as the constants are implicitly public, static, and final. Constants can alternatively be defined using an *enum* declaration.

Protected static fields suffer from the same problem as their public equivalents but also tend to indicate confused design.

Guideline 6-10: Ensure public static final field values are constants

Only immutable values should be stored in public static fields. Many types are mutable and are easily overlooked, in particular arrays and collections. Mutable objects that are stored in a field whose type does not have any mutator methods can be cast back to the runtime type. Enum values should never be mutable.

```
    import static java.util.Arrays.asList;
    import static java.util.Collections.unmodifiableList;
    ...
    public static final List<String> names = unmodifi-
ableList(asList(
        "Fred", "Jim", "Sheila"
        ));
```

As per Guideline 6-10, protected static fields suffer from the same problems astheir public equivalents.

Guideline 6-11: Do not expose mutable statics

Private statics are easily exposed through public interfaces, if sometimes only in a limited way (see Guidelines 6-2 and 6-6). Mutable statics may also change behaviour between unrelated code. To ensure safe code, private statics should be treated as if they are public. Adding boilerplate to expose statics as singletons does not fix these issues.

Mutable statics may be used as caches of immutable flyweight values. Mutable objects should never be cached in statics. Even instance pooling of mutable objects should be treated with extreme caution.

7 Object Construction

During construction objects are at an awkward stage where they exist but are not ready for use. Such awkwardness presents a few more difficulties in addition to those of ordinary methods.

Guideline 7-1: Avoid exposing constructors of sensitive classes

Construction of classes can be more carefully controlled if constructors are not exposed. Define static factory methods instead of public constructors. Support extensibility through delegation rather than inheritance. Implicit constructors through serialization and clone should also be avoided.

Guideline 7-2: Prevent the unauthorized construction of sensitive classes

Where an existing API exposes a security-sensitive constructor, limit the ability to create instances. 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 crefactory method. Also enforce checks at points where an instance of a class can be cre-

ated without the use of a constructor. Specifically, enforce a check inside the `read-Object` 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 7-3: Defend against partially initialized instances of non-final classes

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

From JDK 6 on, construction of a subclassable class can be prevented by throwing an exception before the `Object` constructor completes. To do this, perform the checks in an expression that is evaluated in a call to `this()` or `super()`.

```
        // non-final java.lang.ClassLoader
        public abstract class ClassLoader {
            protected ClassLoader() {
                this(securityManagerCheck());
            }
            private ClassLoader(Void ignored) {
                // ... continue initialization ...
            }
            private static Void securityManagerCheck() {
                SecurityManager security = System.getSecurity-
Manager();
                if (security != null) {
                    security.checkCreateClassLoader();
                }
                return null;
            }
        }
```

For compatibility with older releases, a potential solution involves the use of an *initialized* flag. Set the flag as the last operation in a constructor before returning successfully. All methods providing a gateway to sensitive operations must first consult the flag before proceeding:

```
public abstract class ClassLoader {

    private volatile boolean initialized;

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

        // Last action of constructor.
        this.initialized = true;
    }
    protected final Class defineClass(...) {
        checkInitialized();

        // regular logic follows
        ...
    }

    private void checkInitialized() {
        if (!initialized) {
            throw new SecurityException(
                "NonFinal not initialized"
            );
        }
    }
}
```

Furthermore, any security-sensitive uses of such classes should check the state of the initialization flag. In the case of ClassLoader construction, it should check that its parent class loader is initialized.

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 mali-

cious `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.

A more robust, but also more verbose, approach is to use a "pointer to implementation" (or "pimpl"). The core of the class is moved into a non-public class with the interface class forwarding method calls. Any attempts to use a the class before it is fully initialized will result in a `NullPointerException`. This approach is also good for dealing with clone and deserialization attacks.

```
public abstract class ClassLoader {

    private final ClassLoaderImpl impl;

    protected ClassLoader() {
        this.impl = new ClassLoaderImpl();
    }
    protected final Class defineClass(...) {
        return impl.defineClass(...);
    }
}

/* pp */ class ClassLoaderImpl {
    /* pp */ ClassLoaderImpl() {
        // permission needed to create ClassLoader
        securityManagerCheck();
        init();
    }

    /* pp */ Class defineClass(...) {
```

```
                // regular logic follows
                ...
            }
        }
```

Guideline 7-4: Prevent constructors from calling methods that can be overridden

Constructors that call overridable methods give attackers a reference to `this` (the object being constructed) before the object has been fully initialized. Likewise, `clone`, `readObject`, or `readObjectNoData` methods that call overridable methods may do the same. The `readObject` methods will usually call `java.io.ObjectInputStream.defaultReadObject`, which is an overridable method.

Guideline 7-5: Defend against cloning of non-final classes

A non-final class may be subclassed by a class that also implements `java.lang.Cloneable`. The result is that the base class can be unexpectedly cloned, although only for instances created by an adversary. The clone will be a shallow copy. The twins will share referenced objects but have different fields and separate intrinsic locks. The pointer to implementation approach detailed in Guideline 7-3 provides a good defence.

8 Serialization and Deserialization

Java Serialization provides an interface to classes that sidesteps the normal expectations of the Java language.

Guideline 8-1: Avoid serialization for security-sensitive classes

Security-sensitive classes that are not serializable will not have the problems detailed in this section. Making a class serializable effectively creates a public interface to all fields of that class.

Guideline 8-2: Guard sensitive data during serialization

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

Approaches for handling sensitive fields in serializable classes are:

- Declare sensitive fields `transient`
- Define the `serialPersistentFields` array field appropriately
- Implement `writeObject` and use `ObjectOutputStream.putField` selectively
- Implement `writeReplace` to replace the instance with a serial proxy
- Implement the `Externalizable` interface

Guideline 8-3: View deserialization the same as object construction

Deserialization creates a new instance of a class without invoking any constructor on that class. Therefore, deserialization should be designed to behave to match normal construction.

Default deserialization and `ObjectInputStream.defaultReadObject` can assign arbitrary objects to non-transient fields and does not necessarily return. Use `ObjectInputStream.readFields` instead to insert copying before assignment to fields. Or, if possible, don't make sensitive classes serializable.

```
        public final class ByteString implements java.io.Seri-
alizable {
            private static final long serialVersionUID = 1L;
            private byte[] data;
            public ByteString(byte[] data) {
                this.data = data.clone(); // Make copy before
assignment.
            }
            private void readObject(
                java.io.ObjectInputStream in
            ) throws java.io.IOException, ClassNotFoundExcep-
tion {
                java.io.ObjectInputStreadm.GetField fields =
                    in.readFields();
                this.data =
((byte[])fields.get("data")).clone();
            }
            ...
        }
```

Perform the same input validation checks in a `readObject` method implementation as those performed in a constructor. Likewise, assign default values that are consistent with those assigned in a constructor to all fields, including transient fields, which are 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 deserializing byte streams that are specially crafted to give the attacker references to mutable objects inside the deserialized container object.

```
        public final class Nonnegative implements java.io.Seri-
alizable {
            private static final long serialVersionUID = 1L;
            private int value;
            public Nonnegative(int value) {
                // Make check before assignment.
                this.data = nonnegative(value);
            }
            private static int nonnegative(int value) {
                if (value < 0) {
                    throw new IllegalArgumentException(value +
                                                       " is
negative");
                }
                return value;
            }
            private void readObject(
                java.io.ObjectInputStream in
            ) throws java.io.IOException, ClassNotFoundExcep-
tion {
                java.io.ObjectInputStreadm.GetField fields =
                    in.readFields();
                this.value = nonnegative(field.get(value, 0));
            }
            ...
        }
```

Attackers can also craft hostile streams in an attempt to exploit partially initialized

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, use 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 8-4: 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 Sensitive-
Class
                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
                // 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.Seri-
alizable {

            // 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 setName(String name) {
                if (name!=null ? name.equals(this.name)
                               : (this.name == null)) {
                    // no change - do nothing
                    return;
                } else {
                    // permission needed to modify name
                    securityManagerCheck();

                    inputValidation(name);

                    this.name = name;
                }
            }

            // implement readObject to enforce checks
            //   during deserialization

            private void readObject(java.io.ObjectInputStream
in) {
                java.io.ObjectInputStream.GetField fields =
                    in.readFields();
                String name = (String) fields.get("name", DE-
FAULT);

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


                if (!DEFAULT.equals(name)) {
                    securityManagerCheck();
                    inputValidation(name);
                }
                this.name = 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.Seri-
alizable {
            // sensitive internal state
            private String value;

            // public method to allow callers to retrieve in-
ternal 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);
            }
        }
```

Guideline 8-5: Understand the security permissions given to serialization and deserialization

Permissions appropriate for deserialization should be carefully checked.

Serialization with full permissions allows permission checks in `writeObject` methods to be circumvented. For instance, `java.security.GuardedObject` checks the guard before serializing the target object. With full permissions, this guard can be circumvented and the data from the object (although not the object itself) made available to the attacker.

Deserialization is more significant. A number of `readObject` implementations attempt to make security checks, which will pass if full permissions are granted. Further, some non-serializable security-sensitive, subclassable classes have no-argument constructors, for instance `ClassLoader`. Consider a malicious serializable class that subclasses `ClassLoader`. During deserialization the serialization method calls the constructor itself and then runs any readObject in the subclass. When the `ClassLoader` constructor is called no unprivileged code is on the stack, hence security checks will pass. Thus, don't deserialize with permissions unsuitable for the data.

9 Access Control

Although Java is largely an *object-capability* language, a stack-based access control mechanism is used to securely provide more conventional APIs.

Guideline 9-1: Understand how permissions are checked

The standard security check ensures that each frame in the call stack has the required permission. That is, the current permissions in force is the *intersection* of the permissions of each frame in the current access control context. If any frame does not have a permission, no matter where it lies in the stack, then the current context does not have that permission.

Consider an application that indirectly uses secure operations through a library.

```
        package xx.lib;

        public class LibClass {
            private static final String OPTIONS = "xx.lib.op-
tions";

            public static String getOptions() {
                // checked by SecurityManager
                return System.getProperty(OPTIONS);
            }
        }

        package yy.app;

        class AppClass {
            public static void main(String[] args) {
                System.out.println(
                    xx.lib.LibClass.getOptions()
                );
            }
        }
```

When the permission check is performed, the call stack will be as illustrated below.

```
        +--------------------------------+
        | java.security.AccessController |
        |   .checkPermission(Permission) |
        +--------------------------------+
        | java.lang.SecurityManager      |
        |   .checkPermission(Permission) |
        +--------------------------------+
        | java.lang.SecurityManager      |
        |   .checkPropertyAccess(String) |
        +--------------------------------+
        | java.lang.System               |
        |   .getProperty(String)         |
        +--------------------------------+
        | xx.lib.LibClass                |
        |   .getOptions()                |
        +--------------------------------+
        | yy.app.AppClass                |
        |   .main(String[])              |
        +--------------------------------+
```

In the above example, if the `AppClass` frame does not have permission to read a file but the `LibClass` frame does, then a security exception is still thrown. It does not matter that the immediate caller of the privileged operation is fully privileged, but that there is unprivileged code on the stack somewhere.

For library code to appear transparent to applications with respect to privileges, libraries should be granted permissions at least as generous as the application code that it is used with. For this reason, almost all the code shipped in the JDK and extensions is fully privileged. It is therefore important that there be at least one frame with the application's permissions on the stack whenever a library executes security checked operations on behalf of application code.

Guideline 9-2: Beware of callback methods

Callback methods are generally invoked from the system with full permissions. It seems reasonable to expect that malicious code needs to be on the stack in order to perform an operation, but that is not the case. Malicious code may set up objects that bridge the callback to a security checked operation. For instance, a file chooser dialog box that can manipulate the filesystem from user actions, may have events posted from malicious code. Alternatively, malicious code can disguise a file chooser as something benign while redirecting user events.

Callbacks are widespread in object-oriented systems. Examples include the following:

- Static initialization is often done with full privileges
- Application main method
- Applet/Midlet/Servlet lifecycle events
- `Runnable.run`

This bridging between callback and security-sensitive operations is particularly tricky because it is not easy to spot the bug or to work out where it is.

When implementing callback types, use the technique described in Guideline 9-6 to transfer context. For instance, `java.beans.EventHandler` is a dynamic proxy that may be used to implement callback interfaces. `EventHandler` captures the context when it is constructed and uses that to execute the target operation. This means that instances of `EventHandler` may well be security-sensitive.

Guideline 9-3: Safely invoke `java.security.AccessController.doPrivileged`

`AccessController.doPrivileged` enables code to exercise its own permissions when performing `SecurityManager`-checked operations. For the purposes of security checks, the call stack is effectively truncated below the caller of `doPrivileged`. The immediate caller is included in security checks.

```
+--------------------------------+
| action                         |
|   .run                         |
+--------------------------------+
| java.security.AccessController |
|   .doPrivileged                |
+--------------------------------+
| SomeClass                      |
|   .someMethod                  |
+--------------------------------+
| ~~OtherClass~~                     |
|   ~~.otherMethod~~                 |

+--------------------------------+
        |                               |
```

In the above example, the privileges of the `OtherClass` frame are ignored for security checks.

To avoid inadvertently performing such operations on behalf of unauthorized callers, be very careful when invoking `doPrivileged` using caller-provided inputs (tainted inputs):

```
        package xx.lib;

        import java.security.*;

        public class LibClass {
            private static final String OPTIONS = "xx.lib.op-
tions";

            public static String getOptions() {
                return AccessController.doPrivileged(
                    new PrivilegedAction<String>() {
                        public String run() {
                            // this is checked by SecurityMan-
ager
                            return System.getProperty(OPTIONS);
                        }
                    }
                );
            }
        }
```

The implementation of `getOptions` properly retrieves the system property using a hardcoded value. More specifically, it does not allow the caller to influence the name of the property 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.

Privileged code sections should be made as small as practical in order to make comprehension of the security implications tractable.

By convention, instances of `PrivilegedAction` and `PrivilegedExceptionAction` may be made available to untrusted code, but `doPrivileged` must not be invoked with caller-provided actions.

The two-argument overloads of `doPrivileged` allow changing of privileges to that of a previous acquired context. A null context is interpreted as adding no further restrictions. Therefore, before using stored contexts, make sure that they are not `null` (`AccessController.getContext` never returns `null`).

```
        if (acc == null) {
            throw new SecurityException("Missing AccessControl-
Context");
        }
        AccessController.doPrivileged(new
PrivilegedAction<Void>() {
                public Void run() {
                    ...
                }
            }, acc);
```

Guideline 9-4: Know how to restrict privileges through `doPrivileged`

As permissions are restricted to the intersection of frames, an artificial `AccessControlContext` representing no (zero) frames implies all permissions. The following three calls to `doPrivileged` are equivalent:

```
        private static final AccessControlContext allPermis-
sionsAcc =
            new AccessControlContext(
                new java.security.ProtectionDomain[0]
            );
        void someMethod(PrivilegedAction<Void> action) {
            AccessController.doPrivileged(action, allPermis-
sionsAcc);
            AccessController.doPrivileged(action, null);
            AccessController.doPrivileged(action);
        }
```

All permissions can be removed using an artificial `AccessControlContext` context containing a frame of a `ProtectionDomain` with no permissions:

```
        private static final java.security.PermissionCollection
            noPermissions = new java.security.Permissions();
        private static final AccessControlContext noPermission-
sAcc =
            new AccessControlContext(
                new ProtectionDomain[] {
                    new ProtectionDomain(null, noPermissions)
                }
            );

        void someMethod(PrivilegedAction<Void> action) {
            AccessController.doPrivileged(new PrivilegedAc-
tion<Void>() {
                public Void run() {
                    ... context has no permissions ...
                    return null;
                }
            }, noPermissionsAcc);
        }

        +--------------------------------+
        | ActionImpl                     |
        |   .run                         |
        +--------------------------------+
        |                                |
        | noPermissionsAcc               |
        + - - - - - - - - - - - - - - - -+
        | java.security.AccessController |
        |   .doPrivileged                |
        +--------------------------------+
        | SomeClass                      |
        |   .someMethod                  |
        +--------------------------------+

        |  ~~OtherClass~~                    |
        |    ~~.otherMethod~~                |
        +--------------------------------+

        |                                |
```

An intermediate situation is possible where only a limited set of permissions is granted. If the permissions are checked in the current context before being supplied to `doPrivileged`, permissions may be reduced without the risk of privilege elevation. This enables the use of the principle of least privilege:

```
        private static void doWithFile(final Runnable task,
                                       String knownPath) {
            Permission perm = new java.io.FilePermission(known-
Path,

"read,write");

            // Ensure context already has permission,
            //   so privileges are not elevate.
            AccessController.checkPermission(perm);

            // Execute task with the single permission only.
            PermissionCollection perms = perm.newPermissionCol-
lection();
            perms.add(perm);
            AccessController.doPrivileged(new PrivilegedAc-
tion<Void>() {
                public Void run() {
                    task.run();
                    return null;
                }},
                new AccessControlContext(
                    new ProtectionDomain[] {
                        new ProtectionDomain(null, perms)
                    }
                )
            );
        }
```

Guideline 9-5: Be careful caching results of potentially privileged operations

A cached result must never be passed to a context that does not have the relevant permissions to generate it. Therefore, ensure that the result is generated in a context that has no more permissions than any context it is returned to. Because calculation of privileges may contain errors, use the AccessController API to enforce the constraint.

```
        private static final Map<String> cache;

        public static Thing getThing(String key) {
            // Try cache.
            CacheEntry entry = cache.get(key);
            if (entry != null) {
                // Ensure we have required permissions before
returning
                //   cached result.
                AccessController.checkPermission(entry.getPer-
mission());
                return entry.getValue();
            }

            // Ensure we do not elevate privileges (per Guide-
line 9-2).
            Permission perm = getPermission(key);
            AccessController.checkPermission(perm);

            // Create new value with exact privileges.
            PermissionCollection perms = perm.newPermissionCol-
lection();
            perms.add(perm);
            Thing value = AccessController.doPrivileged(
                new PrivilegedAction<Thing>() { public Thing
run() {
                    return createThing(key);
                }},
                new AccessControlContext(
                    new ProtectionDomain[] {
                        new ProtectionDomain(null, perms)
                    }
                `
                ,
            );
            cache.put(key, new CacheEntry(value, perm));

            return value;
        }
```

Guideline 9-6: Understand how to transfer context

It is often useful to store an access control context for later use. For example, one may decide it is appropriate to provide access to callback instances that perform privileged operations, but invoke callback methods in the context that the callback object was registered. The context may be restored later on in the same thread or in a different thread. A particular context may be restored multiple times and even after the original thread has exited.

`AccessController.getContext` returns the current context. The two-argument forms of `AccessController.doPrivileged` can then replace the current context with the stored context for the duration of an action.

```
        package xx.lib;

        public class Reactor {
            public void addHandler(Handler handler) {
                handlers.add(new HandlerEntry(
                        handler, AccessController.getContext()
                ));
            }
            private void fire(final Handler handler,
                              AccessControlContext acc) {
                if (acc == null) {
                    throw new SecurityException(
                                  "Missing AccessControlCon-
text");
                }
                AccessController.doPrivileged(
                    new PrivilegedAction<Void>() {
                        public Void run() {
                            handler.handle();
                            return null;
                        }

                    }, acc);
             }
             ...
        }


                                         +---------------------
-----------+
                                         | xx.lib.FileHandler
|
                                         |   handle()
|
                                         +---------------------
-----------+
                                         |
xx.lib.Reactor.(anonymous)     |
                                         |   run()
|
+--------------------------------+ \     +---------------------
-----------+
| java.security.AccessController |  `    |
|
|   .getContext()                |  +--> | acc
|
+--------------------------------+  |    + - - - - - - - - - - -
- - - - - -+
| xx.lib.Reactor                 |  |    | java.security.Ac-
cessController |
|   .addHandler(Handler)         |  |    |   .doPrivileged(han-
dler, acc)  |
+--------------------------------+  |    +---------------------
-----------+
| yy.app.App                     |  |    | xx.lib.Reactor
|
|   .main(String[] args)         |  ,    |   .fire
|
+--------------------------------+ /     +---------------------
-----------+
-----------.
                                                  | ~~xx.lib.Reactor~~
|
                                                  | ~~.run~~
|
                                                  +--------------------
-----------+
                                                  |
|
```

Guideline 9-7: Understand how thread construction transfers context

Newly constructed threads are executed with the access control context that was present when the `Thread` object was constructed. In order to prevent bypassing this context, `void run()` of untrusted objects should not be executed with inappropriate privileges.

Guideline 9-8: 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 [1] 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 that they do not inadvertently invoke `Class.newInstance` on behalf of untrusted code.

```
        package yy.app;

        class AppClass {
           OtherClass appMethod() throws Exception {
               return OtherClass.class.newInstance();
           }
        }

        +--------------------------------+
        | xx.lib.LibClass                |
        |   .LibClass                    |
        +--------------------------------+
        | java.lang.Class                |
        |   .newInstance                 |
        +--------------------------------+
        | yy.app.AppClass                |<--
AppClass.class.getClassLoader
        |   .appMethod                   |       determines
check
        +--------------------------------+

        |                                |
```

Code has full access to its own class loader and any class loader that is a descendent. In the case of `Class.newInstance` access to a class loader implies access to classes in restricted packages (e.g., sun.* in the Sun JDK).

In the diagram below, classes loaded by B have access to B and its descendents C, E, and F. Other class loaders, shown in grey strikeout font, are subject to security checks.

```
        +-------------------------+

        |    ~~bootstrap loader~~     | <--- null
        +-------------------------+
                ^              ^
        +------------------+  +---+
        | ~~extension loader~~ |  | ~~A~~ |
        +------------------+  +---+
                 ^
        +------------------+

        |  ~~system loader~~    | <--- Class.getSystemClassLoader()
        +------------------+
             ^           ^
        +----------+   +---+

        |    B     |   | ~~F~~ |
        +----------+   +---+
          ^      ^       ^


        +---+  +---+   +---+
        | C |  | E |   | ~~G~~ |
        +---+  +---+   +---+
          ^
        +---+
        | D |
        +---+
```

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
```

```
        java.lang.Thread.getContextClassLoader
```

Refrain from invoking the above methods on Class, ClassLoader, or Thread instances that are 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 that are returned by the above methods back to untrusted code.

Guideline 9-9: 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 that are returned by these methods back to untrusted code.

Guideline 9-10: Be aware of standard APIs that perform Java language access checks against the immediate caller

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

Code may also call standard APIs (primarily in the `java.lang.reflect` package) to reflectively access fields or methods in another object. The following reflectionbased APIs mirror the language checks that are 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.newUp-
dater

java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater
        java.util.concurrent.atomic.AtomicReferenceFieldUp-
dater.newUpdater
```

Language checks are performed solely against the immediate caller, not against each caller in the execution sequence. Because 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 that are received from untrusted code. If the respective instances were acquired safely, do not invoke the above methods using inputs that are provided by untrusted code. Also, do not propagate objects that are returned by the above methods back to untrusted code.

The `java.beans` package provides safe alternatives to some of these methods.

Guideline 9-11: Be aware `java.lang.reflect.Method.invoke` is ignored for checking the immediate caller

Consider:

```
        package xx.lib;

        class LibClass {
            void libMethod(
                PrivilegedAction action
            ) throws Exception {
                Method doPrivilegedMethod =
                    AccessController.class.getMethod(
                        "doPrivileged", PrivilegedAction.class
                    );
                doPrivilegedMethod.invoke(null, action);
            }
        }
```

If `Method.invoke` was taken as the immediate caller, then the action would be performed with all permissions. So, for the methods discussed in Guidelines 9-8 through 9-10, the `Method.invoke` implementation is ignored when determining the immediate caller.

```
        +--------------------------------+
        | action                         |
        |   .run                         |
        +--------------------------------+
        | java.security.AccessController |
        |   .doPrivileged                |
        +--------------------------------+

        | ~~java.lang.reflect.Method~~       |
        |    ~~.invoke~~                     |
        +--------------------------------+
        | xx.lib.LibClass                | <--- Effective
caller
        |   .libMethod                   |

        +--------------------------------+
        |                                |
```

Therefore, avoid `Method.invoke.` Use `java.beans.Statement/Expression` to provide a safer approach to similar functionality.

Conclusion

The Java Platform provides a robust basis for secure systems through features such as memory-safety. However, the platform alone cannot prevent flaws being introduced. This document details many of the common pitfalls. The most effective approach to minimizing vulnerabilities is to have obviously no flaws rather than no obvious flaws.

References

1. Li Gong, Gary Ellison, and Mary Dageforde.
   Inside Java 2 Platform Security. 2nd ed.
   Boston, MA: Addison-Wesley, 2003.
2. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.
   The Java Language Specification.
   3rd ed. Boston, MA: Addison-Wesley, 2005.
3. Tim Lindholm and Frank Yellin.
   The Java Virtual Machine Specification. 2nd ed.
   Reading, MA: Addison-Wesley, 1999.
4. Aleph One. Smashing the Stack for Fun and Profit.
   Phrack 49, November 1996.
5. John Viega and Gary McGraw.
   Building Secure Software: How to Avoid Security Problems the Right Way.
   Boston: Addison-Wesley, 2002.
6. Joshua Bloch. Effective Java Programming Language Guide.
   2nd ed. Addison-Wesley Professional, 2008.
7. Gary McGraw. Software Security: Building Security In.
   Boston: Addison-Wesley, 2006.
8. C.A.R. Hoare. The Emperor's Old Clothes.
   Communications of the ACM, 1981