Secure Coding Guidelines for Java SE

Updated for Java SE 13
Document version: 7.3
Published: 27 September 2018
Last updated: 23 September 2019

Introduction

One of the main design considerations for the Java platform is to provide a restricted environment for executing code with different permission levels. Java comes with its own unique set of security challenges. While the Java security architecture [1] can in many cases help to protect users and systems from hostile or misbehaving code, 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. In severe cases local programs may be executed or Java security disabled. These bugs can potentially be used to steal confidential data from the machine and intranet, misuse system resources, 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 highly resistant to the stack-smashing [4] and buffer overflow attacks possible in the C and to a lesser extent C++ programming languages. The explicit static typing of Java makes code easy to understand (and facilitates static analysis), and the dynamic checks ensure unexpected conditions result in predictable behavior.

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 document bridges such publications together and includes coverage of additional topics. It 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, 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.

While sections 0 through 3 are generally applicable across different types of software, most of the guidelines in sections 4 through 9 focus on applications that interact with untrusted code (though some guidelines in these sections are still relevant for other situations). Developers should analyze the interactions that occur across an application's trust boundaries and identify the types of data involved to determine which guidelines are relevant. Performing threat modeling and establishing trust boundaries can help to accomplish this (see Guideline 0-4).

These guidelines are intended to help developers build secure software, but they do not focus specifically on software that implements security features. Therefore, topics such as cryptography are not covered in this document (see [9] and [10] for information on using cryptography with Java). While adding features to software can solve some security-related problems, it should not be relied upon to eliminate security defects.

This document has been updated to cover some of the new features included in Java SE 11. However, these guidelines are also applicable to software written for previous versions of Java.

0 Fundamentals

The following general principles apply throughout Java security.

Guideline 0-0 / FUNDAMENTALS-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 / FUNDAMENTALS-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 / FUNDAMENTALS-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 / FUNDAMENTALS-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 9).

Rich Internet Applications (RIA) can specify their requested permissions in the JNLP. A signed jar can also include a manifest attribute that specifies whether it must run in a sandbox or with all permissions (see [11]). If a sandboxed application attempts to execute security-sensitive code, the JRE will throw a security exception. RIAs should follow the principle of least privilege, and should be configured to run with the least amount of necessary permissions. Running a RIA with all permissions should be avoided whenever possible.

Guideline 0-4 / FUNDAMENTALS-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 audits 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 / FUNDAMENTALS-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. Note, however, that care must be taken by both the code performing the check and the caller to prevent the capability from being leaked to code without the proper permissions. See Section 9 for additional information.

Guideline 0-6 / FUNDAMENTALS-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.

Guideline 0-7 / FUNDAMENTALS-7: Document security-related information

API documentation should cover security-related information such as required permissions, security-related exceptions, caller sensitivity (see Guidelines 9-8 through 9-11 for additional on this topic), and any preconditions or postconditions that are relevant to security. Documenting this information in comments for a tool such as Javadoc can also help to ensure that it is kept up to date.

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. In addition to attacks that cause excessive resource consumption, attacks that result in persistent DoS, such as wasting significant disk space, need be defended against. Server systems should be especially robust against external attacks.

Guideline 1-1 / DOS-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 / DOS-2: Release resources in all cases

Some objects, such as open files, locks and manually allocated memory, behave as resources which require every acquire operation to be paired with a definite release. It is easy to overlook the vast possibilities for executions paths when exceptions are thrown. Resources should always be released promptly no matter what.

Even experienced programmers often handle resources incorrectly. In order to reduce errors, duplication should be minimized and resource handling concerns should be separated. The Execute Around Method pattern provides an excellent way of extracting the paired acquire and release operations. The pattern can be used concisely using the Java SE 8 lambda feature.

```
        long sum = readFileBuffered(InputStream in -> {
            long current = 0;
            for (;;) {
                int b = in.read();
                if (b == -1) {
                    return current;
                }
                current += b;
            }
        });
```

The try-with-resource syntax introduced in Java SE 7 automatically handles the release of many resource types.

```
        public  R readFileBuffered(
            InputStreamHandler handler
        ) throws IOException {
            try (final InputStream in = Files.newInputStream(path)) {
                handler.handle(new BufferedInputStream(in));
            }
        }
```

For 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.

```
        public  R locked(Action action) {
            lock.lock();
            try {
                return action.run();
            } finally {
                lock.unlock();
            }
        }
```

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.

```
        public void writeFile(
            OutputStreamHandler handler
        ) throws IOException {
            try (final OutputStream rawOut = Files.newOutputStream(path)) {
                final BufferedOutputStream out =
                    new BufferedOutputStream(rawOut);
                handler.handle(out);
                out.flush();
            }
        }
```

Some decorators of resources may themselves be resources that require correct release. For instance, in the current Oracle JDK implementation compression-related streams are natively implemented using the C heap for buffer storage. Care must be taken that both resources are released in all circumstances.

```
        public void bufferedWriteGzipFile(
            OutputStreamHandler handler
        ) throws IOException {
            try (
                final OutputStream rawOut = Files.newOutputStream(path);
                final OutputStream compressedOut =
                                        new GzipOutputStream(rawOut);
            ) {
                final BufferedOutputStream out =
                    new BufferedOutputStream(compressedOut);
                handler.handle(out);
                out.flush();
            }
        }
```

Note, however, that in certain situations a try statement may never complete running (either normally or abruptly). For example, code inside of the try statement could indefinitely block while attempting to access a resource. If the try statement calls into untrusted code, that code could also intentionally sleep or block in order to prevent the cleanup code from being reached. As a result, resources used in a try-with-resources statement may not be closed, or code in a finally block may never be executed in these situations.

Guideline 1-3 / DOS-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, some operations on primitive integral 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(current);
            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`.

As of Java SE 8, the `java.lang.Math` class also contains methods for various operations (`addExact`, `multiplyExact`, `decrementExact`, etc.) that throw an `ArithmeticException` if the result overflows the given type.

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 / CONFIDENTIAL-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 sometimes also necessary to sanitize exceptions containing information derived from caller inputs. For example, exceptions related to file access could disclose whether a file exists. An attacker may be able to gather useful information by providing various file names as input and analyzing the resulting exceptions.

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 / CONFIDENTIAL-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 / CONFIDENTIAL-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 / INJECT-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 / INJECT-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.

An example of using PreparedStatement correctly:

```
        String sql = "SELECT * FROM User WHERE userId = ?";
        PreparedStatement stmt = con.prepareStatement(sql);
        stmt.setString(1, userId);
        ResultSet rs = prepStmt.executeQuery();
```

Guideline 3-3 / INJECT-3: XML and HTML generation requires care

Untrusted data should be properly sanitized before being included in HTML or XML output. Failure to properly sanitize the data can lead to many different security problems, such as Cross-Site Scripting (XSS) and XML Injection vulnerabilities. It is important to be particularly careful when using Java Server Pages (JSP).

There are many different ways to sanitize data before including it in output. Characters that are problematic for the specific type of output can be filtered, escaped, or encoded. Alternatively, characters that are known to be safe can be allowed, and everything else can be filtered, escaped, or encoded. This latter approach is preferable, as it does not require identifying and enumerating all characters that could potentially cause problems.

Implementing correct data sanitization and encoding can be tricky and error-prone. Therefore, it is better to use a library to perform these tasks during HTML or XML construction.

Guideline 3-4 / INJECT-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 / INJECT-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 safest way to avoid these problems while maintaining the power of XML is to reduce privileges (as described in Guideline 9-2) and to use the most restrictive configuration possible for the XML parser. Reducing privileges still allows you to grant some access, such as inclusion to pages from the same-origin web site if necessary. XML parsers can also be configured to limit functionality based on what is required, such as disallowing external entities or disabling DTDs altogether.

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

Guideline 3-6 / INJECT-6: Care with BMP files

BMP images files may contain references to local ICC (International Color Consortium) 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 / INJECT-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 / INJECT-8: Take care interpreting untrusted code

Code can be hidden in a number of places. If the source is not trusted to supply code, 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 `javax.script` scripting API or similar.
LiveConnect interfaces with JavaScript running in the browser. The JavaScript running on a web page will not usually have been verified with an object code signing certificate.
By default the Oracle 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.
Java Sound will load code through the `javax.sound.midi.MidiSystem.getSoundbank` methods.
RMI may allow loading of remote code specified by remote connection. On the Oracle JDK, this is disabled by default but may be enabled or disabled through the `java.rmi.server.useCodebaseOnly` system property.
LDAP (RFC 2713) allows loading of remote code in a server response. On the Oracle JDK, this is disabled by default but may be enabled or disabled through the `com.sun.jndi.ldap.object.trustURLCodebase` system property.
Many SQL implementations allow execution of code with effects outside of the database itself.

Guideline 3-9 / INJECT-9: Prevent injection of exceptional floating point values

Working with floating point numbers requires care when importing those from outside of a trust boundary, as the NaN (not a number) or infinite values can be injected into applications via untrusted input data, for example by conversion of (untrusted) Strings converted by the `Double.valueOf` method. Unfortunately the processing of exceptional values is typically not immediately noticed without introducing sanitization code. Moreover, passing an exceptional value to an operation propagates the exceptional numeric state to the operation result.

Both positive and negative infinity values are possible outcomes of a floating point operation [2], when results become too high or too low to be representable by the memory area that backs a primitive floating point value. Also, the exceptional value NaN can result from dividing 0.0 by 0.0 or subtracting infinity from infinity.

The results of casting propagated exceptional floating point numbers to short, integer and long primitive values need special care, too. This is because an integer conversion of a NaN value will result in a 0, and a positive infinite value is transformed to `Integer.MAX_VALUE` (or `Integer.MIN_VALUE` for negative infinity), which may not be correct in certain use cases.

There are distinct application scenarios where these exceptional values are expected, such as scientific data analysis which relies on numeric processing. However, it is advised that the result values be contained for that purpose in the local component. This can be achieved by sanitizing any floating point results before passing them back to the generic parts of an application.

As mentioned before, the programmer may wish to include sanitization code for these exceptional values when working with floating point numbers, especially if related to authorization or authentication decisions, or forwarding floating point values to JNI. The `Double` and `Float` classes help with sanitization by providing the `isNan` and `isInfinite` methods. Also keep in mind that comparing instances of `Double.NaN` via the equality operator always results to be false, which may cause lookup problems in maps or collections when using the equality operator on a wrapped double field within the equals method in a class definition.

A typical code pattern that can block further processing of unexpected floating point numbers is shown in the following example snippet.

```
        if (Double.isNaN(untrusted_double_value)) {
            // specific action for non-number case
        }

        if (Double.isInfinite(untrusted_double_value)){
            // specific action for infinite case
        }

        // normal processing starts here
```

4 Accessibility and Extensibility

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

Guideline 4-1 / EXTEND-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 / EXTEND-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 = "package.access";
        static {
            String packageAccess = java.security.Security.getProperty(
                PACKAGE_ACCESS_KEY
            );
            java.security.Security.setProperty(
                PACKAGE_ACCESS_KEY,
                (
                    (packageAccess == null ||
                     packageAccess.trim().isEmpty()) ?
                    "" :
                    (packageAccess + ",")
                ) +
                "xx.example.product.implementation."
            );
        }
```

Guideline 4-3 / EXTEND-3: Isolate unrelated code

Containers, that is to say code that manages code with a lower level of trust, should isolate unrelated application code. Even otherwise untrusted code is typically given permissions to access its origin, and therefore untrusted code from different origins should be isolated.

Although there may be security checks on direct accesses, there are indirect ways of using the system class loader and thread context class loader. Programs should be written with the expectation that the system class loader is accessible everywhere and the thread context class loader is accessible to all code that can execute on the relevant threads.

Mutable statics (see Guideline 6-11) and exceptions are common ways that isolation is inadvertently breached. Mutable statics allow any code to interfere with code that directly or, more likely, indirectly uses them.

Library code can be carefully written such that it is safely usable by less trusted code. Libraries require a level of trust at least equal to the code it is used by in order not to violate the integrity of the client code. Containers should ensure that less trusted code is not able to replace more trusted library code and does not have package-private access. Both restrictions are typically enforced by using a separate class loader instance, the library class loader a parent of the application class loader.

Guideline 4-4 / EXTEND-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 cast 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 / EXTEND-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.getSecurityManager();
                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 / EXTEND-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) // SecurityManager
                                 remove(key)   // checks for these
                                               // methods
```

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 behavior 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.

JDK 8 introduced default methods on interfaces. These default methods are another path for new and unexpected methods to show up in a class. If a class implements an interface with default methods, those are now part of the class and may allow unexpected access to internal data. For a security sensitive class, all interfaces implemented by the class (and all superclasses) would need to be monitored as previously discussed.

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 / INPUT-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 / INPUT-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.

A subtle example would be `Class` objects returned by `ClassLoaders`. An attacker might be able to control `ClassLoader` instances that get passed as arguments, or that are set in `Thread` context. Thus, when calling methods on `ClassLoaders` not many assumptions can be made. Multiple invocations of `ClassLoader.loadClass()` are not guaranteed to return the same `Class` instance or definition, which could cause TOCTOU issues.

Guideline 5-3 / INPUT-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 = Integer.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 native 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.

The examples in this section use `java.util.Date` extensively as it is an example of a mutable API class. In an application, it would be preferable to use the new Java Date and Time API (`java.time.*`) which has been designed to be immutable.

Guideline 6-1 / MUTABLE-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 / MUTABLE-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 / MUTABLE-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(); // HttpCookie 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.getTime()));
    }
    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 / MUTABLE-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 / MUTABLE-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 IdentityHashMap<>();

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 PrivateKey();

    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 / MUTABLE-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 IOException {
    // Copy (clone) private mutable data before sending.
    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 / MUTABLE-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 / MUTABLE-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 / MUTABLE-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 modifiers "`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 / MUTABLE-10: Ensure public static final field values are constants

Only immutable or unmodifiable 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.

In the following example, names exposes an unmodifiable view of a list in order to prevent the list from being modified.

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

The `of()` and `ofEntries()` API methods, which were added in Java 9, can also be used to create unmodifiable collections:

```
        public static final List<String> names =
                                    List.of("Fred", "Jim", "Sheila");
```

Note that the `of/ofEntries` API methods return an unmodifiable collection, whereas the `Collections.unmodifiable...` API methods (`unmodifiableCollection()`, `unmodifiableList()`, `unmodifiableMap()`, etc.) return an unmodifiable view to a collection. While the collection cannot be modified via the unmodifiable view, the underlying collection may still be modified via a direct reference to it. However, the collections returned by the `of/ofEntries` API methods are in fact unmodifiable. See the `java.util.Collections` API documentation for a complete list of methods that return unmodifiable views to collections.

The `copyOf` methods, which were added in Java 10, can be used to create unmodifiable copies of existing collections. Unlike with unmodifiable views, if the original collection is modified the changes will not affect the unmodifiable copy. Similarly, the `toUnmodifiableList()`, `toUnmodifiableSet()`, and `toUnmodifiableMap()` collectors in Java 10 and later can be used to create unmodifiable collections from the elements of a stream.

As per Guideline 6-9, protected static fields suffer from the same problems as their public equivalents.

Guideline 6-11 / MUTABLE-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 behavior 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.

Some mutable statics require a security permission to update state. The updated value will be visible globally. Therefore mutation should be done with extreme care. Methods that update global state or provide a capability to do so, with a security check, include:

```
java.lang.ClassLoader.getSystemClassLoader
java.lang.System.clearProperty
java.lang.System.getProperties
java.lang.System.setErr
java.lang.System.setIn
java.lang.System.setOut
java.lang.System.setProperties
java.lang.System.setProperty
java.lang.System.setSecurityManager
java.net.Authenticator.setDefault
java.net.CookieHandler.getDefault
java.net.CookieHandler.setDefault
java.net.Datagram.setDatagramSocketImplFactory
java.net.HttpURLConnection.setFollowRedirects
java.net.ProxySelector.setDefault
java.net.ResponseCache.getDefault
java.net.ResponseCache.setDefault
java.net.ServerSocket.setSocketFactory
java.net.Socket.setSocketImplFactory
java.net.URL.setURLStreamHandlerFactory
java.net.URLConnection.setContentHandlerFactory
java.net.URLConnection.setFileNameMap
java.rmi.server.RMISocketFactory.setFailureHandler
java.rmi.server.RMISocketFactory.setSocketFactory
java.rmi.activation.ActivationGroup.createGroup
java.rmi.activation.ActivationGroup.setSystem
java.rmi.server.RMIClassLoader.getDefaultProviderInstance
java.security.Policy.setPolicy
java.sql.DriverManager.setLogStream (Deprecated)
java.sql.DriverManager.setLogWriter
java.util.Locale.setDefault
java.util.TimeZone.setDefault
javax.naming.spi.NamingManager.setInitialContextFactoryBuilder
javax.naming.spi.NamingManager.setObjectFactoryBuilder
javax.net.ssl.HttpsURLConnection.setDefaultHostnameVerifier
javax.net.ssl.HttpsURLConnection.setDefaultSSLSocketFactory
javax.net.ssl.SSLContext.setDefault
javax.security.auth.login.Configuration.setConfiguration
javax.security.auth.login.Policy.setPolicy
```

Java PlugIn and Java WebStart isolate certain global state within an `AppContext`. Often no security permissions are necessary to access this state, so it cannot be trusted (other than for Same Origin Policy within PlugIn and WebStart). While there are security checks, the state is still intended to remain within the context. Objects retrieved directly or indirectly from the `AppContext` should therefore not be stored in other variations of globals, such as plain statics of classes in a shared class loader. Any library code directly or indirectly using `AppContext` on behalf of an application should be clearly documented. Users of `AppContext` include:

```
Extensively within AWT
Extensively within Swing
Extensively within JavaBeans Long Term Persistence
java.beans.Beans.setDesignTime
java.beans.Beans.setGuiAvailable
java.beans.Introspector.getBeanInfo
java.beans.PropertyEditorFinder.registerEditor
java.beans.PropertyEditorFinder.setEdiorSearchPath
javax.imageio.ImageIO.createImageInputStream
javax.imageio.ImageIO.createImageOutputStream
javax.imageio.ImageIO.getUseCache
javax.imageio.ImageIO.setCacheDirectory
javax.imageio.ImageIO.setUseCache
javax.print.StreamPrintServiceFactory.lookupStreamPrintServices
javax.print.PrintServiceLookup.lookupDefaultPrintService
javax.print.PrintServiceLookup.lookupMultiDocPrintServices
javax.print.PrintServiceLookup.lookupPrintServices
javax.print.PrintServiceLookup.registerService
javax.print.PrintServiceLookup.registerServiceProvider
```

Guideline 6-12 / MUTABLE-12: Do not expose modifiable collections

Classes that expose collections either through public variables or get methods have the potential for side effects, where calling classes can modify contents of the collection. Developers should consider exposing read-only copies of collections relating to security authentication or internal state.

While modification of a field referencing a collection object can be prevented by declaring it `final` (see Guideline 6-9), the collection itself must be made unmodifiable separately. An unmodifiable collection can be created using the `of/ofEntries` API methods (available in Java 9 and later), or the `copyOf` API methods (available in Java 10 and later). An unmodifiable view of a collection can be obtained using the `Collections.unmodifiable...` APIs.

In the following example, an unmodifiable collection is exposed via `SIMPLE`, and unmodifiable views to modifiable collections are exposed via `ITEMS` and `somethingStateful`.

```
public class Example {
    public static final List<String> SIMPLE =
        List.of("first", "second", "...");
    public static final Map<String, String> ITEMS;

    static {
        //For complex items requiring construction
        Map<String, String> temp = new HashMap<>(2);
        temp.put("first", "The first object");
        temp.put("second", "Another object");
        ITEMS = Collections.unmodifiableMap(temp);
```

```
            }

            private List<String> somethingStateful =
                                  new ArrayList<SomethingImmutable>();
            public List<String> getSomethingStateful() {
                    return  Collections.unmodifiableList(
                                      somethingStateful);
            }
        }
```

Arrays exposed via public variables or get methods can introduce similar issues. For those cases, a copy of the internal array (created using `clone()`, `java.util.Arrays.copyOf()`, etc.) should be exposed instead.

Note that all of the collections in the previous example contain immutable objects. If a collection or array contains mutable objects, then it is necessary to expose a deep copy of it instead. See Guidelines 6-2 and 6-3 for additional information on creating safe copies.

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 / OBJECT-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 / OBJECT-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 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 7-3 / OBJECT-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.getSecurityManager();
                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 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.

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 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 / OBJECT-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 / OBJECT-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 defense.

8 Serialization and Deserialization

Note: Deserialization of untrusted data is inherently dangerous and should be avoided.

Java Serialization provides an interface to classes that sidesteps the field access control mechanisms of the Java language. As a result, care must be taken when performing serialization and deserialization. Furthermore, deserialization of untrusted data should be avoided whenever possible, and should be performed carefully when it cannot be avoided (see 8-6 for additional information).

Guideline 8-1 / SERIAL-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. Serialization also effectively adds a hidden public constructor to a class, which needs to be considered when trying to restrict object construction.

Similarly, lambdas should be scrutinized before being made serializable. Functional interfaces should not be made serializable without due consideration for what could be exposed.

Guideline 8-2 / SERIAL-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 / SERIAL-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 like 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.Serializable {
    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, ClassNotFoundException {
            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.Serializable {
        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, ClassNotFoundException {
            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 (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.

Security-sensitive serializable classes should ensure that object field types are final classes, or do special validation to ensure exact types when deserializing. Otherwise attacker code may populate the fields with malicious subclasses which behave in unexpected ways. For example, if a class has a field of type `java.util.List`, an attacker may populate the field with an implementation which returns inconsistent data.

Guideline 8-4 / SERIAL-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 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 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", DEFAULT);

            // 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.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);
        }
    }
```

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

Permissions appropriate for deserialization should be carefully checked. Additionally, deserialization of untrusted data should generally be avoided whenever possible.

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. Instead, data should be deserialized with the least necessary privileges.

Guideline 8-6 / SERIAL-6: Filter untrusted serial data

Serialization Filtering was introduced in JDK 9 to improve both security and robustness when using Object Serialization. Security guidelines consistently require that input from external sources be validated before use; serialization filtering provides a mechanism to validate classes before they are deserialized. Filters can be configured that apply to every use of object deserialization without modifying the application. The filters are configured via system properties or configured using the override mechanism of the security properties. A typical use case is to black-list classes that have been identified as potentially compromising the Java runtime. White-listing known safe classes is also straight-forward (and preferred over a black-list approach for stronger security). The filter mechanism allows object-serialization clients to more easily validate their inputs. For a more fine-grained approach the `ObjectInputFilter` API allows an application to integrate finer control specific to each use of `ObjectInputStream`.

RMI supports the setting of a serialization filters to protect remote invocations of exported objects. The RMI Registry and RMI distributed garbage collector use the filtering mechanisms defensively.

Support for the configurable filters has been included in the CPU releases for JDK 8u121, JDK 7u131, and JDK 6u141.

For more information and details please refer to [17] and [20].

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.

Many of the guidelines in this section cover the use of the SecurityManager to perform security checks, and to elevate or restrict permissions for code. Note that the SecurityManager does not and cannot provide protection against issues such as side-channel attacks or lower level problems such as Row hammer, nor can it guarantee complete intra-process isolation. Separate processes (JVMs) should be used to isolate untrusted code from trusted code with sensitive information. Utilizing lower level isolation mechanisms available from operating systems or containers is also recommended.

Guideline 9-1 / ACCESS-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.options";

    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 / ACCESS-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.

Guideline 9-3 / ACCESS-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 {
            // System property used by library,
            //  does not contain sensitive information
            private static final String OPTIONS = "xx.lib.options";

            public static String getOptions() {
                return AccessController.doPrivileged(
                    new PrivilegedAction<String>() {
                        public String run() {
                            // this is checked by SecurityManager
                            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`.

It is also important to ensure that privileged operations do not leak sensitive information. Whenever the return value of `doPrivileged` is made accessible to untrusted code, verify that the returned object does not expose sensitive information. In the above example, `getOptions` returns the value of a system property, but the property does not contain any sensitive data.

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 AccessControlContext");
        }
        AccessController.doPrivileged(new PrivilegedAction<Void>() {
                public Void run() {
                    ...
                }
            }, acc);
```

Guideline 9-4 / ACCESS-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 allPermissionsAcc =
            new AccessControlContext(
                new java.security.ProtectionDomain[0]
            );
        void someMethod(PrivilegedAction<Void> action) {
            AccessController.doPrivileged(action, allPermissionsAcc);
            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 noPermissionsAcc =
            new AccessControlContext(
                new ProtectionDomain[] {
                    new ProtectionDomain(null, noPermissions)
                }
            );

        void someMethod(PrivilegedAction<Void> action) {
            AccessController.doPrivileged(new PrivilegedAction<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(knownPath,
                                                         "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.newPermissionCollection();
            perms.add(perm);
            AccessController.doPrivileged(new PrivilegedAction<Void>() {
                public Void run() {
                    task.run();
                    return null;
                }},
                new AccessControlContext(
                    new ProtectionDomain[] {
                        new ProtectionDomain(null, perms)
                    }
                )
            );
        }
```

When granting permission to a directory, extreme care must be taken to ensure that the access does not have unintended consequences. Files or subdirectories could have insecure permissions, or filesystem objects could provide additional access outside of the directory (e.g. symbolic links, loop devices, network mounts/shares, etc.). It is important to consider this when granting file permissions via a security policy or `AccessController.doPrivileged` block, as well as for less obvious cases (e.g. classes can be granted read permission to the directory from which they were loaded).

Applications should utilize dedicated directories for code as well as for other filesystem use, and should ensure that secure permissions are applied. Running code from or granting access to shared/common directories (including access via symbolic links) should be avoided whenever possible. It is also recommended to configure file permission checking to be as strict and secure as possible [21].

A limited `doPrivileged` approach was also added in Java 8. This approach allows code to assert a subset of its privileges while still allowing a full access-control stack walk to check for other permissions. If a check is made for one of the asserted permissions, then the stack check will stop at the `doPrivileged` invocation. For other permission checks, the stack check continues past the `doPrivileged` invocation. This differs from the previously discussed approach, which will always stop at the `doPrivileged` invocation.

Consider the following example:

```
        private static void doWithURL(final Runnable task,
                                              String knownURL) {
            URLPermission urlPerm = new URLPermission(knownURL);
            AccessController.doPrivileged(
                                  new PrivilegedAction<Void>() {
                public Void run() {
                    task.run();
                    return null;
                }},
                someContext,
                urlPerm
            );
        }
```

If a permission check matching the `URLPermission` is performed during the execution of task, then the stack check will stop at `doWithURL`. However, if a permission check is performed that does not match the `URLPermission` then the stack check will continue to walk the stack.

As with other versions of `doPrivileged`, the context argument can be null with the limited `doPrivileged` methods, which results in no additional restrictions being applied.

Guideline 9-5 / ACCESS-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 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.getPermission());
                return entry.getValue();
            }

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

            // Create new value with exact privileges.
            PermissionCollection perms = perm.newPermissionCollection();
            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 / ACCESS-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 AccessControlContext");
        }
        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.AccessController |
|   .addHandler(Handler)         |  |    |   .doPrivileged(handler, acc)  |
+--------------------------------+  |    +--------------------------------+
| yy.app.App                     |  |    | xx.lib.Reactor                 |
|   .main(String[] args)         |  ,    |   .fire                        |
+--------------------------------+ /     +--------------------------------+
                                         | ~~xx.lib.Reactor~~         |
                                         | ~~.run~~                   |
                                         +--------------------------------+
                                         |                                |
```

Guideline 9-7 / ACCESS-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 / ACCESS-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.getClassLoader` 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 `getClassLoader` 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.getClassLoader` 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.getClassLoader` on behalf of untrusted code.

```
package yy.app;

class AppClass {
   ClassLoader appMethod() throws Exception {
       return OtherClass.class.getClassLoader();
   }
}
+--------------------------------+
| xx.lib.LibClass                |
|   .LibClass                    |
+--------------------------------+
| java.lang.Class                |
|   .getClassLoader                 |
+--------------------------------+
| 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 descendant. In the case of `Class.getClassLoader` access to a class loader implies access to classes in restricted packages (e.g., system classes prefixed with `sun.`).

In the diagram below, classes loaded by B have access to B and its descendants C, E, and D. 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.getClassLoader`, and potentially bypass `SecurityManager` checks depending on the immediate caller's class loader:

```
java.io.ObjectStreamField.getType
java.io.ObjectStreamClass.forClass
java.lang.Class.newInstance (Deprecated)
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.Class.getDeclaringClass
java.lang.Class.getEnclosingMethod
java.lang.Class.getEnclosingClass
java.lang.Class.getEnclosingConstructor
java.lang.Class.getNestHost
java.lang.Class.getNestMembers
java.lang.ClassLoader.getParent
java.lang.ClassLoader.getSystemClassLoader
java.lang.StackWalker.forEach
java.lang.StackWalker.getCallerClass
java.lang.StackWalker.walk
java.lang.invoke.MethodHandleProxies.asInterfaceInstance
java.lang.reflect.Proxy.getInvocationHandler
java.lang.reflect.Proxy.getProxyClass
java.lang.reflect.Proxy.newProxyInstance
java.lang.Thread.getContextClassLoader
javax.sql.rowset.serial.SerialJavaObject.getFields
```

Methods such as these that vary their behavior according to the immediate caller's class are considered to be caller-sensitive, and should be annotated in code with the @CallerSensitive annotation [16].

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 / ACCESS-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 (Deprecated)
java.lang.ClassLoader.getPlatformClassLoader
java.lang.Package.getPackage(s)
java.lang.Runtime.load
java.lang.Runtime.loadLibrary
java.lang.System.load
java.lang.System.loadLibrary
java.sql.DriverManager.deregisterDriver
java.sql.DriverManager.getConnection
java.sql.DriverManager.getDriver(s)
java.security.AccessController.doPrivileged*
java.util.logging.Logger.getAnonymousLogger
java.util.logging.Logger.getLogger
java.util.ResourceBundle.getBundle
```

Methods such as these that vary their behavior according to the immediate caller's class are considered to be caller-sensitive, and should be annotated in code with the @CallerSensitive annotation [16].

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

When an object accesses fields or methods of another object, the JVM performs access control checks to assert the valid visibility of the target method or field. 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 reflection-based APIs mirror the language checks that are enforced by the virtual machine:

```
java.lang.Class.newInstance (Deprecated)
java.lang.invoke.MethodHandles.lookup
java.lang.reflect.AccessibleObject.canAccess
java.lang.reflect.AccessibleObject.setAccessible
java.lang.reflect.AccessibleObject.tryAccessible
java.lang.reflect.Constructor.newInstance
java.lang.reflect.Constructor.setAccessible
java.lang.reflect.Field.get*
java.lang.reflect.Field.set*
java.lang.reflect.Method.invoke
java.lang.reflect.Method.setAccessible
java.util.concurrent.atomic.AtomicIntegerFieldUpdater.newUpdater
java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater
java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater
```

Methods such as these that vary their behavior according to the immediate caller's class are considered to be caller-sensitive, and should be annotated in code with the @CallerSensitive annotation [16].

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.

Guideline 9-11 / ACCESS-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`.

Guideline 9-12 / ACCESS-12: Avoid using caller-sensitive method names in interface classes

When designing an interface class, one should avoid using methods with the same name and signature of caller-sensitive methods, such as those listed in Guidelines 9-8, 9-9, and 9-10. In particular, avoid calling these from default methods on an interface class.

Guideline 9-13 / ACCESS-13: Avoid returning the results of privileged operations

Care should be taken when designing lambdas which are to be returned to untrusted code; especially ones that include security-related operations. Without proper precautions, e.g., input and output validation, untrusted code may be able to leverage the privileges of a lambda inappropriately.

Similarly, care should be taken before returning `Method` objects, `MethodHandle` objects, `MethodHandles.Lookup` objects, `VarHandle` objects, and `StackWalker` objects (depends on options used at creation time) to untrusted code. These objects have checks for language access and/or privileges inherent in their creation and incautious distribution may allow untrusted code to bypass private / protected access restrictions as well as restricted package access. If one returns a `Method` or `MethodHandle` object that an untrusted user would not normally have access to, then a careful analysis is required to ensure that the object does not convey undesirable capabilities. Similarly, `MethodHandles.Lookup` objects have different capabilities depending on who created them. It is important to understand the access granted by any such object before it is returned to untrusted code.

Guideline 9-14 / ACCESS-14: Safely invoke standard APIs that perform tasks using the immediate caller's module

The following static methods perform tasks using the immediate caller's `Module`:

```
        java.lang.System.getLogger
        java.lang.Class.forName(String, Module)
        java.lang.Class.getResourceAsStream
        java.lang.Class.getResource
        java.lang.Module.addExports
        java.lang.Module.addOpens
        java.lang.Module.addReads
        java.lang.Module.addUses
        java.lang.Module.getResourceAsStream
        java.lang.Module.getResource
        java.util.ResourceBundle.getBundle
        java.util.ServiceLoader.load
        java.util.ServiceLoader.loadInstalled
```

Methods such as these that vary their behavior according to the immediate caller's class are considered to be caller-sensitive, and should be annotated in code with the @CallerSensitive annotation [16].

For example, `Module::addExports` uses the immediate caller's `Module` to decide if a package should be exported. Do not invoke these methods on behalf of untrusted code, since untrusted code may not have the ability to make the same change.

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 9-15 / ACCESS-15: Design and use InvocationHandlers conservatively

When creating a `java.lang.reflect.Proxy` instance, a class that implements `java.lang.reflect.InvocationHandler` is required to handle the delegation of the methods on the `Proxy` instance. The `InvocationHandler` is assumed to have the permissions of the code that created the `Proxy`. Thus, access to `InvocationHandlers` should not be generally available.

`InvocationHandlers` should also validate the method names they are asked to invoke to prevent the `InvocationHandler` from being used for a purpose for which it was not intended. For example:

```
    public Object invoke(Object proxy, Method method, Object[] args)
                                                 throws Throwable {
       if (! Proxy.isProxyClass(proxy.getClass())) {
            throw new IllegalArgumentException("not a proxy");
        }

        if (Proxy.getInvocationHandler(proxy) != this) {
            throw new IllegalArgumentException("handler mismatch");
        }

        String methodName = method.getName();
        Class[] paramTypes = method.getParameterTypes();
        int paramsLen = paramTypes.length;

        if ( methodName.equals("hashCode") && paramsLen == 0 ) {
            // handle hashCode here
            return true;
        } else ...
            // equals, toString
            // ignore clone, finalize
        } else ...
            // check for methods expected on interfaces implemented
            // method name, number of parameters and types
        }
    }
```

Guideline 9-16 / ACCESS-16: Plan module configuration carefully

When planning a module's configuration, use strong encapsulation to limit the exported packages (both qualified and unqualified exports). Examine all exported packages/classes to be sure that nothing security sensitive has been exposed. Exporting additional packages in the future is easy but rescinding an export could cause compatibility issues.

There are command line options to open / export specific packages beyond what the module configuration specifies. Minimizing the need for their usage is also recommended.

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, Gilad Bracha, and Alex Buckley.
   The Java Language Specification, Java SE 13 Edition.
   http://docs.oracle.com/javase/specs/
3. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
   The Java Virtual Machine Specification, Java SE 13 Edition.
   http://docs.oracle.com/javase/specs/
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.
   3rd ed. Addison-Wesley Professional, 2018.
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
9. Java SE 13 Security Documentation
10. Trail: Security Features in Java SE
11. JAR File Manifest Attributes for Security
12. Java Native Interface APIs and Developer Guides
13. Sheng Liang.
    The Java Native Interface: Programmer's Guide and Specification.
    Boston: Addison-Wesley, 1999.
14. BlueHat v8: Mitigations Unplugged (Microsoft.com)
15. Security Resource Center
16. JEP 176: Mechanical Checking of Caller-Sensitive Method
17. JEP 290: Filter Incoming Serialization Data
18. 64-bit programming for Game Developers
19. Critical Patch Updates and Security Alerts
20. Serialization Filtering
21. FilePermission API

Appendix A: Defensive use of the Java Native Interface (JNI)

The Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding a JVM into native applications [12] [13]. Native interfaces allow Java programs to interact with APIs that originally do not provide Java bindings. JNI supports implementing these wrappers in C, C++ or assembler. During runtime native methods defined in a dynamically loaded library are connected to a Java method declaration with the native keyword.

JNI-1: Only use JNI when necessary

The easiest security measure for JNI to remember is to avoid native code whenever possible. Therefore, the first task is to identify an alternative that is implemented in Java before choosing JNI as an implementation framework. This is mainly because the development chain of writing, deploying, and maintaining native libraries will burden the entire development chain for the lifetime of the component. Attackers like native code, mainly because JNI security falls back to the security of C/C++, therefore they expect it to break first when attacking a complex application. While it may not always be possible to avoid implementing native code, it should still be kept as short as possible to minimize the attack surface.

JNI-2: Be aware of the C/C++ threat model

Although is it is not impossible to find exploitable holes in the Java layer, C/C++ coding flaws may provide attackers with a faster path towards exploitability. Native antipatterns enable memory exploits (such as heap and stack buffer overflows), but the Java runtime environment safely manages memory and performs automatic checks on access within array bounds. Furthermore, Java has no explicit pointer arithmetic. Native code requires dealing with heap resources carefully, which means that operations to allocate and free native memory require symmetry to prevent memory leaks. Proper heap management during runtime can be checked dynamically with heap checking tools. Depending on the runtime OS platform there may be different offerings (such as valgrind, guardmalloc or pageheap).

JNI-3: Expect that JNI code can violate Java visibility and isolation rules

The Java runtime environment often executes untrusted code, and protection against access to unauthorized resources is a built in feature. In C/C++, private resources such as files (containing passwords and private keys), system memory (private fields) and sockets are essentially just a pointer away. Existing code may contain vulnerabilities that could be instrumented by an attacker (on older operating systems simple shellcode injection was sufficient, whereas advanced memory protections would require the attacker to apply return-oriented programming techniques). This means that C/C++ code, once successfully loaded, is not limited by the Java security policy or any visibility rules.

JNI-4: Secure your JNI implementation from the Java side

In order to prevent native code from being exposed to untrusted and unvalidated data, Java code should sanitize data before passing it to JNI methods. This is also important for application scenarios that process untrusted persistent data, such as deserialization code.

As stated in Guideline 5-3, native methods should be private and should only be accessed through Java-based wrapper methods. This allows for parameters to be validated by Java code before they are passed to native code. The following example illustrates how to validate a pair of offset and length values that are used when accessing a byte buffer. The Java-based wrapper method validates the values and checks for integer overflow before passing the values to a 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 = Integer.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 native code to
        //   crash.

        // will throw ArithmeticException on overflow
        int test = Math.addExact(offset, len);

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

        nativeOperation(data, offset, len);

    }
}
```

While this limits the propagation of maliciously crafted input which an attacker may use to overwrite native buffers, more aspects of the interaction between Java and JNI code require special care. Java hides memory management details like the heap object allocation via encapsulation, but the handling of native objects requires the knowledge of their absolute memory addresses.

To prevent malicious code from misusing operations on native objects to overwrite parts of memory, native operations should be designed without maintaining state. Stateless interaction may not always be possible. To prevent manipulation, native memory addresses kept on the Java side should be kept in private fields and treated as read-only from the Java side. Additionally, references to native memory should never be made accessible to untrusted code.

JNI-5: Properly test JNI code for concurrent access

Especially when maintaining state, be careful testing your JNI implementation so that it behaves stably in multi-threaded scenarios. Apply proper synchronization (prefer atomics to locks, minimize critical sections) to avoid race conditions when calling into the native layer. Concurrency-unaware code will cause memory corruption and other state inconsistency issues in and around the shared data sections.

JNI-6: Secure library loading

The `System.loadLibrary("/com/foo/MyLib.so")` method uses the immediate caller's class loader to find and load the specified native library. Loading libraries enables a caller to invoke native methods. Therefore, do not invoke `loadLibrary` in a trusted library on behalf of untrusted code, since untrusted code may not have the ability to load the same library using its own class loader instance (see Guidelines 9-8 and 9-9 for additional information). Avoid placing a `loadLibrary` call in a privileged block, as this would allow untrusted callers to directly trigger native library initializations. Instead, require a policy with the `loadLibrary` permission granted. As mentioned earlier, parameter validation should also be performed, and `loadLibrary` should not be invoked using input provided by untrusted code. Objects that are returned by native methods should not be handed back to untrusted code.

JNI-7: Perform input validation at the language boundary

To provide in-depth protection against security issues with native memory access, the input passed from the Java layer requires revalidation on the native side. Using the runtime option `-Xcheck:jni` can be helpful catching those issues, which can be fatal to an application, such as passing illegal references, or mixing up array types with non-array types. This option will not protect against subtle semantic conversion errors that can occur on the boundary between native code and Java.

Since values in C/C++ can be unsigned, the native side should check for primitive parameters (especially array indices) to block negative values. Java code is also well protected against type-confusion. However, only a small number of types exist on the native side, and all user objects will be represented by instances of the `jobject` type.

JNI-8: Expect and handle exceptions when calling from JNI into Java

Exceptions are an important construct of the Java language, because they help to distinguish between the normal control flow and any exceptional conditions that can occur during processing. This allows Java code to be prepared for conditions that cause failure.

Native code has no direct support for Java exceptions, and any exceptions thrown by Java code will not affect the control flow of native code. Therefore, native code needs to explicitly check for exceptions after operations, especially when calling into Java methods that may throw exceptions. Exceptions may occur asynchronously, so it is necessary to check for exceptions in long native loops, especially when calling back into Java code.

Be aware that many JNI API methods (e.g. `GetFieldID`) can return NULL or an error code when an exception is thrown. Native code frequently needs to return error values and the calling Java method should be prepared to handle such error conditions accordingly.

Unexpected input and error conditions may cause native code to behave unpredictably. An input whitelist limits the exposure of JNI code to a set of expected values.

JNI-9: Follow secure development practices for the native target platform

Modern operating systems provide a wide range of mechanisms that protect against the exploitability of common native programming bugs, such as stack buffer overflows and the various types of heap corruptions. Stack cookies protect against targeted overwrite of return addresses on the stack, which an attacker could otherwise use to divert control flow. Address Space Layout Randomization prevents attackers from placing formerly well-known return addresses on the stack, which when returning from a subroutine call systems code such as execve on the attacker's behalf. With the above protections, attackers may still choose to place native code snippets (shellcode) within the data heap, an attack vector that is prevented when the operating system allows to flag a memory page as Non-executable (NX).

When building native libraries, some of the above techniques may not be enabled by default and may require an explicit opt-in by the library bootstrap code. In either case it is crucial to know and understand the secure development practice for a given operating system, and adapt the compile and build scripts accordingly [14].

Additionally, 32-/64-bit compatibility recommendations should be followed for the given operating system (e.g. [18]). Whenever possible, pure 64-bit builds should be used instead of relying on compatibility layers such as WOW.

JNI-10: Ensure that bundled JVMs and JREs meet Java's secure baselines

Native applications may contain bundled JVMs and JREs for a variety of purposes. These may expose vulnerabilities over time due to software decay.

System Administrators are responsible for running Java applications in a secure manner, following principle of least privilege, and staying up to date with Java's secure baseline (either for standard Java SE or the Server JRE). Developers of applications containing an embedded JVM/JRE should also provide application updates to apply security fixes to the JVM/JRE.

Previously unknown attack vectors can be eliminated by regularly updating the JRE/JDK with critical patch updates from Oracle Technical Network [19]. To keep updates as easy as possible vendors should minimize or even better avoid customization of files in the JRE directory.

Java SDKs and Tools

- Java SE
- Java EE and Glassfish
- Java ME
- Java Card
- NetBeans IDE
- Java Mission Control

Java Resources

- Java APIs
- Technical Articles
- Demos and Videos
- Forums
- Java Magazine
- Developer Training
- Tutorials
- Java.com

E-mail this page Printer View