TEMENOS
The Banking Software Company

PRODUCT

Secure Coding Standards for JAVA

Amendment History

Version Number	Date	Author/Changed by	Description of Changes Made
0.1			Draft
0.2	18-Oct-12	Smitha Nair	Formatting done for the existing document
1.0	19-Oct-12	EPG Subgroup	Baselined for publishing.

Contents

1. Introduction

This document takes the main points of the Java secure coding standards that apply to jBC. This document needs to be reworked to provide proper jBC examples where necessary.

It has been structured in the same sections as the Java Coding standards.

2. Fundamentals

2.1 Guideline 2-1: Prefer to have obviously no flaws rather than no obvious flaws

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.

2.2 Guideline 2-2: 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. Any use of the permissions checking highlights an area that should be scrutinized.

2.3 Guideline 2-3: 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.

2.4 Guideline 2-4: 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.

2.5 Guideline 2-5: Establish trust boundaries

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

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

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

2.6 Guideline 2-6: Minimise the number of permission checks

Permissions checks should be considered a last resort. Perform security checks at a few defined points and return capability data that client code retains so that no further permission checks are required.

2.7 Guideline 2-7: Encapsulate

Allocate behaviors and provide succinct interfaces. Fields of objects should not be exposed and accessors avoided. The interface of a <method, class, package, and module> should form a coherent set of behaviors, and no more.

3. Denial of Service

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

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

3.1 Guideline 3-1: 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.
- 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.

3.2 Guideline 3-2: Release resources in all cases

Exceptions are often overlooked. Resources should always be released promptly even if an exception is thrown.

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.

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

4. 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.

4.1 Guideline 4-1: Purge sensitive information from exceptions

Error messages may convey sensitive information. For example, if a routine accesses a configuration file and that file is not present the details of the file could be in an underlying error message. Propagating this message back to the 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.

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.

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

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

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

4.2 Guideline 4-2: 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.

4.3 Guideline 4-3: 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.

5. 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.

5.1 Guideline 5-1: 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.

5.2 Guideline 5-2: 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.

5.3 Guideline 5-3: XML and HTML generation requires care

Cross Site Scripting (XSS) is a common vulnerability in web applications. This is generally caused by inputs included in outputs without first validating the input. For example, checking

for illegal characters and escaping data properly. It is better to use a library that constructs XML or HTML rather than attempting to insert escape codes in every field in every document. In particular, be careful when using Java Server Pages (JSP).

5.4 Guideline 5-4: 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.

5.5 Guideline 5-5: Restrict XML inclusion

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

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

5.6 Guideline 5-6: Care with BMP files

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

5.7 Guideline 5-7: Disable HTML display in Swing components

Not Applicable

5.8 Guideline 5-8: Take care interpreting untrusted code

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

6. Accessibility and Extensibility

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

6.1 Guideline 6-1: 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.

6.2 Guideline 6-2: 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."
    );
}
```

6.3 Guideline 6-3: Isolate unrelated code

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

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

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

6.4 Guideline 6-4: Limit exposure of ClassLoader instances

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

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

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

7. 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.

7.1 Guideline 7-1: 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).

7.2 Guideline 7-2: 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.

7.3 Guideline 7-3: 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);
    }
}
```

8. Mutability

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

8.1 Guideline 8-1: 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]).

8.2 Guideline 8-2: 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();
    }
}
```

8.3 Guideline 8-3: 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.

8.4 Guideline 8-4: 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.

8.5 Guideline 8-5: 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 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);
        ...
    }
}
```

8.6 Guideline 8-6: 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 {
    out.write(data.clone()); // Copy private mutable data before sending.
}
```

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.

8.7 Guideline 8-7: 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;
}
```

8.8 Guideline 8-8: 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.

8.9 Guideline 8-9: Make public static fields final

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

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

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

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

8.10 Guideline 8-10: Ensure public static final field values are constants

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

```
import static java.util.Arrays.asList;
import static java.util.Collections.unmodifiableList;
...
public static final List<String> names = unmodifiableList(asList(
```

```
        "Fred", "Jim", "Sheila"
    ));
```

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

8.11 Guideline 8-11: Do not expose mutable statics

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

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

9. Access Control

9.1 Guideline 9-1: Understand how permissions are checked

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

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.

9.2 Guideline 9-2: 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.

9.3 Guideline 9-3: 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.