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.