JEP 415: Context-Specific Deserialization Filters

Owner	Roger Riggs
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	17
Component	core-libs / java.io:serialization
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	S
Duration	S
Relates to	JEP 290: Filter Incoming Serialization Data
Reviewed by	Brian Goetz, Chris Hegarty
Endorsed by	Brian Goetz
Created	2021/03/10 15:36
Updated	2022/04/08 14:05
Issue	8263381

Summary

Allow applications to configure context-specific and dynamically-selected deserialization filters via a JVM-wide filter factory that is invoked to select a filter for each individual deserialization operation.

Non-Goals

- It is not a goal to define policies for deserialization filter selection.
- It is not a goal to define a mechanism for the configuration or distribution of filters.

Motivation

Deserializing untrusted data is an inherently dangerous activity because the content of the incoming data stream determines the objects that are created, the values of their fields, and the references between them. In many typical uses the bytes in the stream are received from an unknown, untrusted, or unauthenticated client. By careful construction of the stream, an adversary can cause code in arbitrary classes to be executed with malicious intent. If object construction has side effects that change state or invoke other actions, those actions can compromise the integrity of application objects, library objects, and even the Java runtime. The key to disabling deserialization attacks is to prevent instances of arbitrary classes from being deserialized, thereby preventing the direct or indirect execution of their methods.

We introduced deserialization filters (JEP 290) in Java 9 to enable application and library code to validate incoming data streams before deserializing them. Such code supplies validation logic as a `java.io.ObjectInputFilter` when it creates a deserialization stream (i.e., a `java.io.ObjectInputStream`).

Relying on a stream's creator to explicitly request validation has several limitations. This approach does not scale, and makes it difficult to update filters after code has been shipped. It also cannot impose filtering on deserialization operations performed by third-party libraries in an application.

To address these limitations, JEP 290 also introduced a JVM-wide deserialization filter which can be set via an API, system properties, or security properties. This filter is *static* since it is specified exactly once, at startup. Experience with the static JVM-wide filter has revealed that it, too, has limitations, particularly in complex applications with layers of libraries and multiple execution contexts. Using the JVM-wide filter for every `ObjectInputStream` requires the filter to cover every execution context in the application, so the filter usually winds up being either too inclusive or too restrictive.

A better approach would be to configure per-stream filters in a way that does not require the participation of every stream creator.

To protect the JVM against deserialization vulnerabilities, application developers need a clear description of the objects that can be serialized or deserialized by each component or library. For each context and use case, developers should construct and apply an appropriate filter. For example, if the application uses a specific library to deserialize a particular cohort of objects then a filter for the relevant classes can be applied when calling the library. Creating an allow-list of classes, and rejecting everything else, gives protection against objects in a stream that are otherwise unknown or unexpected. Encapsulation or other natural application or library partitioning boundaries can be used to narrow the set of objects that are allowed or definitely not allowed. If it is not practical to have an allow-list then a reject-list should include classes, packages, and modules that are known not to occur in the stream or are known to be malicious.

An application’s developer is in the best position to understand the structure and operation of the application’s components. This enhancement enables the application developer to construct and apply filters to every deserialization operation.

Description

As noted above, JEP 290 introduced both per-stream deserialization filters and a static JVM-wide filter. Whenever an `ObjectInputStream` is created, its per-stream filter is initialized to be the static JVM-wide filter. That per-stream filter can later be changed to a different filter, if desired.

Here we introduce a configurable JVM-wide *filter factory*. Whenever an `ObjectInputStream` is created, its per-stream filter is initialized to the value returned by invoking the static JVM-wide filter factory. Thus these filters are *dynamic* and *context-specific*, unlike the single static JVM-wide deserialization filter. For backward compatibility, if a filter factory is not set then a built-in factory returns the static JVM-wide filter if one was configured.

The filter factory is used for every deserialization operation in the Java runtime, whether in application code, library code, or code in the JDK itself. The factory is specific to the application and should take into account every deserialization execution context within the application. The filter factory is called from the `ObjectInputStream` constructor and also from `ObjectInputStream.setObjectInputFilter`. The arguments are the current filter and a new filter. When called from the constructor, the current filter is `null` and the new filter is the static JVM-wide filter. The factory determines and returns the initial filter for the stream. The factory can create a composite filter with other context-specific controls or just return the static JVM-wide filter. If `ObjectInputStream.setObjectInputFilter` is called, the factory is called a second time with the filter returned from the first call and the requested new filter. The factory determines how to combine the two filters and returns the filter, replacing the filter on the stream.

For simple cases, the filter factory can return a fixed filter for the entire application. For example, here is a filter that allows example classes, allows classes in the `java.base` module, and rejects all other classes:

```
var filter = ObjectInputFilter.Config.createFilter("example.*;java.base/*;!*")
```

In an application with multiple execution contexts, the filter factory can better protect individual contexts by providing a custom filter for each. When the stream is constructed, the filter factory can identify the execution context based upon the current thread-local state, hierarchy of callers, library, module, and class loader. At that point, a policy for creating or selecting filters can choose a specific filter or composition of filters based on the context.

If multiple filters are present then their results can be combined. A useful way to combine filters is to reject deserialization if any of the filters reject it, allow it if any filter allows it, and otherwise remain undecided.

Command Line Use

The properties `jdk.serialFilter` and `jdk.serialFilterFactory` can be set on the command line to set the filter and filter factory. The existing `jdk.serialFilter` property sets a pattern based filter.

The `jdk.serialFilterFactory` property is the class name of the filter factory to be set before the first deserialization. The class must be public and accessible to the application class loader.

For compatibility with JEP 290, if `jdk.serialFilterFactory` property is not set, the filter factory is set to a builtin that provides compatibility with earlier versions.

API

We define two methods in the `ObjectInputFilter.Config` class to set and get the JVM-wide filter factory. The filter factory is a function with two arguments, a current filter and a next filter, and it returns a filter.

```
/**
 * Return the JVM-wide deserialization filter factory.
 *
 * @return the JVM-wide serialization filter factory; non-null
 */
public static BinaryOperator<ObjectInputFilter> getSerialFilterFactory();

/**
 * Set the JVM-wide deserialization filter factory.
 *
 * The filter factory is a function of two parameters, the current filter
 * and the next filter, that returns the filter to be used for the stream.
 *
 * @param filterFactory the serialization filter factory to set as the
 * JVM-wide filter factory; not null
 */
public static void setSerialFilterFactory(BinaryOperator<ObjectInputFilter> filterFactory);
```

Example

This class shows how to filter to every deserialization operation that takes place in the current thread. It defines a thread-local variable to hold the per-thread filter, defines a filter factory to return that filter, configures the factory as the JVM-wide filter factory, and provides a utility function to run a `Runnable` in the context of a specific per-thread filter.

```
public class FilterInThread implements BinaryOperator<ObjectInputFilter> {

    // ThreadLocal to hold the serial filter to be applied
    private final ThreadLocal<ObjectInputFilter> filterThreadLocal = new ThreadLocal<>();

    // Construct a FilterInThread deserialization filter factory.
    public FilterInThread() {}

    /**
     * The filter factory, which is invoked every time a new ObjectInputStream
     * is created.  If a per-stream filter is already set then it returns a
     * filter that combines the results of invoking each filter.
```

```
     *
     * @param curr the current filter on the stream
     * @param next a per stream filter
     * @return the selected filter
     */
    public ObjectInputFilter apply(ObjectInputFilter curr, ObjectInputFilter next) {
        if (curr == null) {
            // Called from the OIS constructor or perhaps OIS.setObjectInputFilter with no current filter
            var filter = filterThreadLocal.get();
            if (filter != null) {
                // Prepend a filter to assert that all classes have been Allowed or Rejected
                filter = ObjectInputFilter.rejectUndecidedClass(filter);
            }
            if (next != null) {
                // Prepend the next filter to the thread filter, if any
                // Initially this is the static JVM-wide filter passed from the OIS constructor
                // Append the filter to reject all UNDECIDED results
                filter = ObjectInputFilter.merge(next, filter);
                filter = ObjectInputFilter.rejectUndecidedClass(filter);
            }
            return filter;
        } else {
            // Called from OIS.setObjectInputFilter with a current filter and a stream-specific filter.
            // The curr filter already incorporates the thread filter and static JVM-wide filter
            // and rejection of undecided classes
            // If there is a stream-specific filter prepend it and a filter to recheck for undecided
            if (next != null) {
                next = ObjectInputFilter.merge(next, curr);
                next = ObjectInputFilter.rejectUndecidedClass(next);
                return next;
            }
            return curr;
        }
    }

    /**
     * Apply the filter and invoke the runnable.
     *
     * @param filter the serial filter to apply to every deserialization in the thread
     * @param runnable a Runnable to invoke
     */
    public void doWithSerialFilter(ObjectInputFilter filter, Runnable runnable) {
        var prevFilter = filterThreadLocal.get();
        try {
            filterThreadLocal.set(filter);
            runnable.run();
        } finally {
            filterThreadLocal.set(prevFilter);
        }
    }
}
```

If a stream-specific filter was already set with `ObjectInputStream::setObjectFilter` then the filter factory combines that filter with the next filter. If either filter rejects a class then that class is rejected. If either filter allows the class then that class is allowed. Otherwise, the result is undecided.

Here's a simple example of using the `FilterInThread` class:

```
// Create a FilterInThread filter factory and set
var filterInThread = new FilterInThread();
ObjectInputFilter.Config.setSerialFilterFactory(filterInThread);

// Create a filter to allow example.* classes and reject all others
var filter = ObjectInputFilter.Config.createFilter("example.*;java.base/*;!*");
filterInThread.doWithSerialFilter(filter, () -> {
      byte[] bytes = ...;
      var o = deserializeObject(bytes);
});
```

Alternatives

JEP 290 allows filters to be implemented as Java classes, thereby allowing complex logic and context awareness. Context-dependent stream-specific filters could be implemented through the use of a delegating filter that is set on every stream. To determine the filter for the specific stream, it would need to examine its caller and map the caller to a specific filter and then delegate to that filter. However, both code complexity and the overhead of determining the caller would impact performance on every invocation.