Java Platform, Standard Edition Core Libraries

2 Serialization Filtering

You can use the Java serialization filtering mechanism to help prevent deserialization vulnerabilities. You can define pattern-based filters or you can create custom filters.

Topics:

Addressing Deserialization Vulnerabilities

An application that accepts untrusted data and deserializes it is vulnerable to attacks. You can create filters to screen incoming streams of serialized objects before they are deserialized.

An object is serialized when its state is converted to a byte stream. That stream can be sent to a file, to a database, or over a network. A Java object is serializable if its class or any of its superclasses implements either the `java.io.Serializable` interface or the `java.io.Externalizable` subinterface. In the JDK, serialization is used in many areas, including Remote Method Invocation (RMI), custom RMI for interprocess communication (IPC) protocols (such as the Spring HTTP invoker), Java Management Extensions (JMX), and Java Messaging Service (JMS).

An object is deserialized when its serialized form is converted to a copy of the object. It is important to ensure the security of this conversion. Deserialization is code execution, because the `readObject` method of the class that is being deserialized can contain custom code. Serializable classes, also known as "gadget classes", can do arbitrary reflective actions such as create classes and invoke methods on them. If your application deserializes these classes, they can cause a denial of service or remote code execution.

When you create a filter, you can specify which classes are acceptable to an application, and which should be rejected. You can control the object graph size and complexity during deserialization so that the object graph doesn't exceed reasonable limits. Filters can be configured as properties, or implemented programmatically.

Besides creating filters, you can take the following actions to help prevent deserialization vulnerabilities:

- Do not deserialized untrusted data.
- Use SSL to encrypt and authenticate the connections between applications.
- Validate field values before assignment, including checking object invariants by using the `readObject` method.

> **Note:**
> Built-in filters are provided for RMI. However, you should use these built-in filters as starting points only. Configure blacklists and/or extend the whitelist to add additional protection for your application that uses RMI. See Built-in Filters (serialization-filtering1.htm#GUID-80AD4DA1-6AA3-42C0-8172-DECF8FB8A841) .

For more information about these and other strategies, see "Serialization and Deserialization" in Secure Coding Guidelines for Java SE (http://www.oracle.com/technetwork/java/seccodeguide-139067.html) .

Java Serialization Filters

The Java serialization filtering mechanism screens incoming streams of serialized objects to help improve security and robustness. Filters can validate incoming classes before they are deserialized.

As stated in JEP 290, the goals of the Java serialization filtering mechanism are to:

- Provide a way to narrow the classes that can be deserialized down to a context-appropriate set of classes.
- Provide metrics to the filter for graph size and complexity during deserialization to validate normal graph behaviors.
- Allow RMI-exported objects to validate the classes expected in invocations.

You can implement serialization filters in the following ways:

- Pattern-based filters do not require you to modify your application. They consist of a sequence of patterns that are defined in properties, in a configuration file or on the command line. Pattern-based filters can accept or reject specific classes, packages, or modules. They can place limits on array sizes, graph depth, total references, and stream size. A typical use case is to blacklist classes that have been identified as potentially compromising the Java runtime. Pattern-based filters are defined for one application or all applications in a process.
- Custom filters are implemented using the `ObjectInputFilter` API. They allow an application to integrate finer control than pattern-based filters, because they can be specific to each `ObjectInputStream`. Custom filters are set on an individual input stream or on all streams in a process.

The filter mechanism is called for each new object in the stream. If more than one active filter (process-wide filter, application filter, or stream-specific filter) exists, only the most specific filter is called.

In most cases, a custom filter should check if a process-wide filter is set. If one exists, the custom filter should invoke it and use the process-wide filter's result, unless the status is `UNDECIDED`.

Support for serialization filters is included starting with JDK 9, and in Java CPU releases starting with 8u121, 7u131, and 6u141.

Whitelists and Blacklists

Whitelists and blacklists can be implemented using pattern-based filters or custom filters. These lists allow you to take proactive and defensive approaches to protect your applications.

The proactive approach uses whitelists to accept only the classes that are recognized and trusted. You can implement whitelists in your code when you develop your application, or later by defining pattern-based filters. If your application only deals with a small set of classes then this approach can work very well. You can implement whitelists by specifying the classes, packages, or modules that are allowed.

The defensive approach uses blacklists to reject classes that are not trusted. Usually, blacklists are implemented after an attack that reveals that a class is a problem. A class can be added to a blacklist, without a code change, by defining a pattern-based filter.

Creating Pattern-Based Filters

Pattern-based filters are filters that you define without changing your application code. You add process-wide filters in properties files, or application-specific filters on the `java` command line.

A pattern-based filter is a sequence of patterns. Each pattern is matched against the name of a class in the stream or a resource limit. Class-based and resource limit patterns can be combined in one filter string, with each pattern separated by a semicolon (;).

Pattern-based Filter Syntax

When you create a filter that is composed of patterns, use the following guidelines:

- Separate patterns by semicolons. For example:

  ```
  pattern1.*;pattern2.*
  ```

- White space is significant and is considered part of the pattern.

- Put the limits first in the string. They are evaluated first regardless of where they are in the string, so putting them first reinforces the ordering. Otherwise, patterns are evaluated from left to right.

- A class that matches a pattern that is preceded by `!` is rejected. A class that matches a pattern without `!` is accepted. The following filter rejects `pattern1.MyClass` but accepts `pattern2.MyClass`:

  ```
  !pattern1.*;pattern2.*
  ```

- Use the wildcard symbol (`*`) to represent unspecified classes in a pattern as shown in the following examples:
  - To match every class, use `*`

- To match every class in `mypackage`, use `mypackage.*`
- To match every class in `mypackage` and its subpackages, use `mypackage.**`
- To match every class that starts with `text`, use `text*`

If a class doesn't match any filter, then it is accepted. If you want to accept only certain classes, then your filter must reject everything that doesn't match. To reject all classes other than those specified, include `!*` as the last pattern in a class filter.

For a complete description of the syntax for the patterns, see the `conf/security/java.security` file, or see JEP 290 (http://openjdk.java.net/jeps/290) .

Pattern-Based Filter Limitations

Pattern-based filters are used for simple acceptance or rejection. These filters have some limitations. For example:

- Patterns can't allow different sizes of arrays based on the class.
- Patterns can't match classes based on the supertype or interfaces of the class.
- Patterns have no state and can't make choices depending on the earlier classes deserialized in the stream.

Define a Pattern-Based Filter for One Application

You can define a pattern-based filter as a system property for one application. A system property supersedes a Security Property value.

To create a filter that only applies to one application, and only to a single invocation of Java, define the `jdk.serialFilter` system property in the command line.

The following example shows how to limit resource usage for an individual application:

```
java -Djdk.serialFilter=maxarray=100000;maxdepth=20;maxrefs=500
com.example.test.Application
```

Define a Pattern-Based Filter for All Applications in a Process

You can define a pattern-based filter as a Security Property, for all applications in a process. A system property supersedes a Security Property value.

1. Edit the `java.security` properties file.
    - JDK 9 and later: `$JAVA_HOME/conf/security/java.security`
    - JDK 8,7,6: `$JAVA_HOME/lib/security/java.security`
2. Add the pattern to the `jdk.serialFilter` Security Property.

Define a Class Filter

You can create a pattern-based class filter that is applied globally. For example, the pattern might be a class name or a package with wildcard.

In the following example, the filter rejects one class from a package (`!example.somepackage.SomeClass`), and accepts all other classes in the package:

```
jdk.serialFilter=!example.somepackage.SomeClass;example.somepackage.*;
```

The previous example filter accepts all other classes, not just those in `example.somepackage.*`. To reject all other classes, add `!*`:

```
jdk.serialFilter=!example.somepackage.SomeClass;example.somepackage.*;!*
```

Define a Resource Limit Filter

A resource filter limits graph complexity and size. You can create filters for the following parameters to control the resource usage for each application:

- Maximum allowed array size. For example: `maxarray=100000;`
- Maximum depth of a graph. For example: `maxdepth=20;`
- Maximum references in a graph between objects. For example: `maxrefs=500;`
- Maximum number of bytes in a stream. For example: `maxbytes=500000;`

Creating Custom Filters

Custom filters are filters you specify in your application's code. They are set on an individual stream or on all streams in a process. You can implement a custom filter as a pattern, a method, a lambda expression, or a class.

Reading a Stream of Serialized Objects

You can set a custom filter on one `ObjectInputStream`, or, to apply the same filter to every stream, set a process-wide filter. If an `ObjectInputStream` doesn't have a filter defined for it, the process-wide filter is called, if there is one.

While the stream is being decoded, the following actions occur:

- For each new object in the stream, the filter is called before the object is instantiated and deserialized.
- For each class in the stream, the filter is called with the resolved class. It is called separately for each supertype and interface in the stream.
- The filter can examine each class referenced in the stream, including the class of objects to be created, supertypes of those classes, and their interfaces.

- For each array in the stream, whether it is an array of primitives, array of strings, or array of objects, the filter is called with the array class and the array length.
- For each reference to an object already read from the stream, the filter is called so it can check the depth, number of references, and stream length. The depth starts at 1 and increases for each nested object and decreases when each nested call returns.
- The filter is not called for primitives or for `java.lang.String` instances that are encoded concretely in the stream.
- The filter returns a status of accept, reject, or undecided.
- Filter actions are logged if logging is enabled.

Unless a filter rejects the object, the object is accepted.

Setting a Custom Filter for an Individual Stream

You can set a filter on an individual `ObjectInputStream` when the input to the stream is untrusted and the filter has a limited set of classes or constraints to enforce. For example, you could ensure that a stream only contains numbers, strings, and other application-specified types.

A custom filter is set using the `setObjectInputFilter` method. The custom filter must be set before objects are read from the stream.

In the following example, the `setObjectInputFilter`method is invoked with the `dateTimeFilter` method. This filter only accepts classes from the `java.time` package. The `dateTimeFilter` method is defined in a code sample in Setting a Custom Filter as a Method (serialization-filtering1.htm#GUID-0A1D23AB-2F18-4979-9288-9CFEC04F207E__GUID-2D9120BA-1262-4CD0-8C0F-2D7B2FE09DC7) .

```
LocalDateTime readDateTime(InputStream is) throws IOException { try (ObjectInputStream
ois = new ObjectInputStream(is)) { ois.setObjectInputFilter(FilterClass::dateTimeFilter);
return (LocalDateTime) ois.readObject(); } catch (ClassNotFoundException ex) {
IOException ioe = new StreamCorruptedException("class missing"); ioe.initCause(ex); throw
ioe; } }
```

Setting a Process-Wide Custom Filter

You can set a process-wide filter that applies to every use of `ObjectInputStream` unless it is overridden on a specific stream. If you can identify every type and condition that is needed by the entire application, the filter can allow those and reject the rest. Typically, process-wide filters are used to reject specific classes or packages, or to limit array sizes, graph depth, or total graph size.

A process-wide filter is set once using the methods of the `ObjectInputFilter.Config` class. The filter can be an instance of a class, a lambda expression, a method reference, or a pattern.

```
ObjectInputFilter filter = ... ObjectInputFilter.Config.setSerialFilter(filter);
```

In the following example, the process-wide filter is set by using a lambda expression.

```
ObjectInputFilter.Config.setSerialFilter(info -> info.depth() > 10 ? Status.REJECTED :
Status.UNDECIDED);
```

In the following example, the process-wide filter is set by using an instance of a class.

```
ObjectInputFilter.Config.setSerialFilter(FilterClass::dateTimeFilter);
```

Setting a Custom Filter Using a Pattern

A pattern-based custom filter, which is convenient for simple cases, can be created by using the `ObjectInputFilter.Config.createFilter` method. You can create a pattern-based filter as a system property or Security Property. Implementing a pattern-based filter as a method or a lambda expression gives you more flexibility.

The filter patterns can accept or reject specific classes, packages, modules, and can place limits on array sizes, graph depth, total references, and stream size. Patterns cannot match the supertype or interfaces of the class.

In the following example, the filter allows `example.File` and rejects `example.Directory` classes.

```
ObjectInputFilter filesOnlyFilter =
ObjectInputFilter.Config.createFilter("example.File;!example.Directory");
```

This example allows only `example.File`. All other classes are rejected.

```
ObjectInputFilter filesOnlyFilter =
ObjectInputFilter.Config.createFilter("example.File;!*");
```

Setting a Custom Filter as a Class

A custom filter can be implemented as a class implementing the `java.io.ObjectInputFilter` interface, as a lambda expression, or as a method.

A filter is typically stateless and performs checks solely on the input parameters. However, you may implement a filter that, for example, maintains state between calls to the `checkInput` method to count artifacts in the stream.

In the following example, the `FilterNumber` class allows any object that is an instance of the `Number` class and rejects all others.

```
class FilterNumber implements ObjectInputFilter { public Status checkInput(FilterInfo
filterInfo) { Class<?> clazz = filterInfo.serialClass(); if (clazz != null) { return
(Number.class.isAssignableFrom(clazz)) ? Status.ALLOWED : Status.REJECTED; } return
Status.UNDECIDED; } }
```

In the example:

- The `checkInput` method accepts an `ObjectInputFilter.FilterInfo` object. The object's methods provide access to the class to be checked, array size, current depth, number of references to existing objects, and stream size read so far.
- If `serialClass` is not null, indicating that a new object is being created, the value is checked to see if the class of the object is `Number`. If so, it is accepted, otherwise it is rejected.
- Any other combination of arguments returns `UNDECIDED`. Deserialization continues, and any remaining filters are run until the object is accepted or rejected. If there are no other filters, the object is accepted.

Setting a Custom Filter as a Method

A custom filter can also be implemented as a method. The method reference is used instead of an inline lambda expression.

The `dateTimeFilter` method that is defined in the following example is used by the code sample in Setting a Custom Filter for an Individual Stream (serialization-filtering1.htm#GUID-0A1D23AB-2F18-4979-9288-9CFEC04F207E__SETTINGAFILTERFORANINDIVIDUALSTREAM-51012200) .

```
public class FilterClass { static ObjectInputFilter.Status
dateTimeFilter(ObjectInputFilter.FilterInfo info) { Class<?> serialClass =
info.serialClass(); if (serialClass != null) { return
serialClass.getPackageName().equals("java.time") ? ObjectInputFilter.Status.ALLOWED :
ObjectInputFilter.Status.REJECTED; } return ObjectInputFilter.Status.UNDECIDED; } }
```

Example: Filter for Classes in the java.base Module

This custom filter, which is also implemented as a method, allows only the classes found in the base module of the JDK. This example works with JDK 9 and later.

```
static ObjectInputFilter.Status baseFilter(ObjectInputFilter.FilterInfo info) { Class<?>
serialClass = info.serialClass(); if (serialClass != null) { return
serialClass.getModule().getName().equals("java.base") ? ObjectInputFilter.Status.ALLOWED
: ObjectInputFilter.Status.REJECTED; } return ObjectInputFilter.Status.UNDECIDED; }
```

Built-in Filters

The Java Remote Method Invocation (RMI) Registry, the RMI Distributed Garbage Collector, and Java Management Extensions (JMX) all have filters that are included in the JDK. You should specify your own filters for the RMI Registry and the RMI Distributed Garbage Collector to add additional protection.

Filters for RMI Registry

Note:

> Use these built-in filters as starting points only. Edit the `sun.rmi.registry.registryFilter` system property to configure blacklists and/or extend the whitelist to add additional protection for the RMI Registry. To protect the whole application, add the patterns to the `jdk.serialFilter` global system property to increase protection for other serialization users that do not have their own custom filters.

The RMI Registry has a built-in whitelist filter that allows objects to be bound in the registry. It includes instances of the `java.rmi.Remote`, `java.lang.Number`, `java.lang.reflect.Proxy`, `java.rmi.server.UnicastRef`, `java.rmi.activation.ActivationId`, `java.rmi.server.UID`, `java.rmi.server.RMIClientSocketFactory`, and `java.rmi.server.RMIServerSocketFactory` classes.

The built-in filter includes size limits:

```
maxarray=1000000,maxdepth=20
```

Supersede the built-in filter by defining a filter using the `sun.rmi.registry.registryFilter` system property with a pattern. If the filter that you define either accepts classes passed to the filter, or rejects classes or sizes, the built-in filter is not invoked. If your filter does not accept or reject anything, the built-filter is invoked.

Filters for RMI Distributed Garbage Collector

> **Note:**
>
> Use these built-in filters as starting points only. Edit the `sun.rmi.transport.dgcFilter` system property to configure blacklists and/or extend the whitelist to add additional protection for Distributed Garbage Collector. To protect the whole application, add the patterns to the `jdk.serialFilter` global system property to increase protection for other serialization users that do not have their own custom filters.

The RMI Distributed Garbage Collector has a built-in whitelist filter that accepts a limited set of classes. It includes instances of the `java.rmi.server.ObjID`, `java.rmi.server.UID`, `java.rmi.dgc.VMID`, and `java.rmi.dgc.Lease` classes.

The built-in filter includes size limits:

```
maxarray=1000000,maxdepth=20
```

Supersede the built-in filter by defining a filter using the `sun.rmi.transport.dgcFilter` system property with a pattern. If the filter accepts classes passed to the filter, or rejects classes or sizes, the built-in filter is not invoked. If the superseding filter does not accept or reject anything, the built-filter is invoked.

Filters for JMX

You can specify the deserialization filter pattern strings to be used while making an `RMIServer.newClient` remote call and while sending deserializing parameters over RMI to the server. You can also provide a filter pattern string to the default agent by using the `management.properties` file.

Logging Filter Actions

You can turn on logging to record the initialization, rejections, and acceptances of calls to serialization filters. Use the log output as a diagnostic tool to see what's being deserialized, and to confirm your settings when you configure whitelists and blacklists.

When logging is enabled, filter actions are logged to the `java.io.serialization` logger.

To enable serialization filter logging, edit the `$JDK_HOME/conf/logging.properties` file.

To log calls that are rejected, add

```
java.io.serialization.level = FINER
```

To log all filter results, add

```
java.io.serialization.level = FINEST
```