JEP 411: Deprecate the Security Manager for Removal

Owner	Sean Mullan
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	17
Component	security-libs / java.security
Discussion	security dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	Alan Bateman, Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2021/04/05 14:19
Updated	2022/07/26 13:38
Issue	8264713

Summary

Deprecate the Security Manager for removal in a future release. The Security Manager dates from Java 1.0. It has not been the primary means of securing client-side Java code for many years, and it has rarely been used to secure server-side code. To move Java forward, we intend to deprecate the Security Manager for removal in concert with the legacy Applet API (JEP 398).

Goals

- Prepare developers for the removal of the Security Manager in a future version of Java.
- Warn users if their Java applications rely on the Security Manager.
- Evaluate whether new APIs or mechanisms are needed to address specific narrow use cases for which the Security Manager has been employed, such as blocking `System::exit`.

Non-Goals

It is not a goal to provide a replacement for the Security Manager. Future JEPs or enhancements may define new APIs or mechanisms for specific use cases, depending upon demand.

Motivation

The Java Platform emphasizes security. The *integrity* of data is protected by the Java language and VM's built-in memory safety: Variables are initialized before use, array bounds are checked, and memory deallocation is completely automatic. Meanwhile, the *confidentiality* of data is protected by the Java class libraries' trusted implementations of modern cryptographic algorithms and protocols such as SHA-3, EdDSA, and TLS 1.3. Security is a dynamic science, thus we continuously update the Java Platform to address new vulnerabilities and to reflect new industry postures, for example by deprecating weak cryptographic protocols.

One long-time element of security is the Security Manager, which dates from Java 1.0. In the era of Java applets downloaded by web browsers, the Security Manager protected the integrity of users' machines and the confidentiality of their data by running applets in a *sandbox*, which denied access to resources such as the file system or the network. The small size of the Java class libraries — only eight `java.*` packages in Java 1.0 — made it feasible for code in, e.g., `java.io` to consult with the Security Manager before performing any operation. The Security Manager drew a bright line between *untrusted code* (applets from a remote machine) and *trusted code* (classes on the local machine): It would approve all operations involving resource access for trusted code but reject them for untrusted code.

As interest in Java grew, we introduced *signed applets* to allow the Security Manager to place trust in remote code, thereby allowing applets to access the same resources as local code run via `java` on the command line. Simultaneously, the Java class libraries were expanding rapidly — Java 1.1 introduced JavaBeans, JDBC, Reflection, RMI, and Serialization — which meant that trusted code gained access to significant new resources such as database connections, RMI servers, and reflective objects. Allowing all trusted code to access all resources was undesirable, so in Java 1.2 we redesigned the Security Manager to focus on applying the *principle of least privilege:* All code would be treated as untrusted by default, subject to sandbox-style controls that prevented access to resources, and users would place trust in specific codebases by granting them specific permissions to access specific resources. In theory, an application JAR on the class path might be more limited in how it uses the JDK than an applet from the Internet. Limiting permissions was seen as a way to constrain the impact of any vulnerabilities that might exist in a body of code — in effect, a defense-in-depth mechanism.

The Security Manager, then, had ambitions to protect against two kinds of threat: *Malicious intent*, especially in remote code, and *accidental vulnerabilities*, especially in local code.

The threat of malicious intent by remote code has receded because the Java Platform no longer supports applets. The Applet API was deprecated in Java 9 in 2017, then deprecated for removal in Java 17 in 2021 with the intent to remove it in a future release. The closed-source browser plugin that ran applets was removed from Oracle's JDK 11 in 2018 along with the closed-source Java Web Start technology. Accordingly, many of the risks that the Security Manager protects against are no longer significant. Furthermore, the Security Manager cannot protect against many risks that now are significant. The Security Manager cannot address 19 of the 25 most dangerous issues identified by industry leaders in 2020, so issues such as XML external entity reference (XXE) injection and improper input

validation have required direct countermeasures in the Java class libraries. (For example, JAXP can protect against XXE attacks and XML entity expansion, while serialization filtering can prevent malicious data from being deserialized before it can do any damage.) The Security Manager is also incapable of preventing malicious behavior based on speculative-execution vulnerabilities.

The Security Manager's lack of efficacy against malicious intent is unfortunate because the Security Manager is, of necessity, woven into the fabric of the Java class libraries. As such, it is an ongoing maintenance burden. All new features and APIs must be evaluated to ensure that they behave correctly when the Security Manager is enabled. Access control based on the least-privilege principle may have been feasible in the class libraries of Java 1.0, but the rapid growth of `java.*` and `javax.*` packages led to dozens of permissions and hundreds of permission checks throughout the JDK. This is a significant surface area to keep secure, especially since permissions can interact in surprising ways. Some permissions, e.g., allow application or library code to perform a series of safe operations whose overall effect is sufficiently unsafe that it would require a more powerful permission if granted directly.

The threat of accidental vulnerabilities in local code is almost impossible to address with the Security Manager. Many of the claims that the Security Manager is widely used to secure local code do not stand up to scrutiny; it is used far less in production than many people assume. There are many reasons for its lack of use:

- **Brittle permission model** — An application developer wishing to benefit from the Security Manager must carefully grant all of the permissions that the application requires for all of the operations that it performs. There is no way to have partial security, where only a few resources are subject to access control. For example, suppose a developer is concerned with illegitimate access to data and thus wishes to grant permission to read files only from a particular directory. Granting the file-read permission is insufficient because the application will almost certainly use other operations in the Java class libraries besides reading files (for example, writing files), and those other operations will be rejected by the Security Manager since the code will not have the appropriate permission. Only a developer who carefully documents how their code interacts with security-sensitive operations in the Java class libraries will be able to grant the necessary permissions. This is not a common developer workflow. (The Security Manager does not allow negative permissions that could express "grant permission for all operations except reading files".)
- **Difficult programming model** — The Security Manager approves a security-sensitive operation by checking the permissions of all running code that led up to the operation. This makes it difficult to write libraries that run with the Security Manager because it is not sufficient for a library developer to document the permissions that their library code needs. It is also necessary for an application developer who uses the library to grant those same permissions to their application code, in addition to any permissions already granted to that code. This violates the principle of least privilege, since the application code may not need the library's permissions for its own operations. The library developer can mitigate this viral growth of permissions by carefully using the `java.security.AccessController` API to request that the Security Manager take only the library's permissions into account, but the complexity of this and other secure coding guidelines is far beyond the interest of most developers. The path of least resistance for application developers is often to grant `AllPermission` to any relevant JAR file, but this again runs counter to the principle of least privilege.
- **Poor performance** — The heart of the Security Manager is a complex access-control algorithm which often imposes an unacceptable performance penalty. For this reason, the Security Manager has always been disabled by default for JVMs run on the command line. This further reduces developers' interest in investing to make libraries and applications run with the Security Manager. The lack of tooling to help infer and validate permissions is an additional impediment.

In the quarter-century since the Security Manager was introduced, adoption has been low. Only a handful of applications ship with policy files that constrain their own operations (e.g., ElasticSearch). Similarly, only a handful of frameworks ship with policy files (e.g., Tomcat), and developers building applications with those frameworks still face the practically insurmountable challenge of figuring out the permissions needed by their own code and by the libraries they use. Some frameworks (e.g., NetBeans) eschew policy files and instead implement a custom Security Manager in order to prevent plugins from calling `System::exit` or to gain insight into code's behavior, such as whether it opens files and network connections — use cases which we think are better served by other means.

In summary, there is no significant interest in developing modern Java applications with the Security Manager. Making access-control decisions based on permissions is unwieldy, slow, and falling out of favor across the industry; .NET, e.g., no longer supports it. Security is better achieved by providing integrity at lower levels of the Java Platform — by, for example, strengthening module boundaries (JEP 403) to prevent access to JDK implementation details, and hardening the implementation itself — and by isolating the entire Java runtime from sensitive resources via out-of-process mechanisms such as containers and hypervisors. To move the Java Platform forward, we will deprecate the legacy Security Manager technology for removal from the JDK. We plan to deprecate and attenuate the capabilities of the Security Manager over a number of releases, simultaneously creating alternative APIs for such tasks as blocking `System::exit` and other use cases considered important enough to have replacements.

Description

In Java 17, we will:

- Deprecate, for removal, most Security Manager related classes and methods.
- Issue a warning message at startup if the Security Manager is enabled on the command line.
- Issue a warning message at run time if a Java application or library installs a Security Manager dynamically.

In Java 18, we will prevent a Java application or library from dynamically installing a Security Manager unless the end user has explicitly opted to allow it. Historically, a Java application or library was always allowed to dynamically install a Security Manager, but since Java 12, the end user has been able to prevent it by setting the system property `java.security.manager` to `disallow` on the command line (`java -Djava.security.manager=disallow ...`) -- this causes `System::setSecurityManager` to throw an `UnsupportedOperationException`. Starting in Java 18, the default value of `java.security.manager` will be `disallow` if not otherwise set via `java -D...`. As a result, applications and libraries that call `System::setSecurityManager` may fail due to an unexpected `UnsupportedOperationException`. In order for `System::setSecurityManager` to work as before, the end user will have to set `java.security.manager` to `allow` on the command line (`java -Djava.security.manager=allow ...`).

In feature releases after Java 18, we will degrade other Security Manager APIs so that they remain in place but with limited or no functionality. For example, we may revise `AccessController::doPrivileged` simply to run the given action, or revise `System::getSecurityManager` always to return `null`. This will allow libraries that support the Security Manager and were compiled against previous Java releases to continue to work without change or even recompilation. We expect to remove the APIs once the compatibility risk of doing so declines to an acceptable level.

In feature releases after Java 18, we may alter the Java SE API definition so that operations which previously performed permission checks no longer perform them, or perform fewer checks when a Security Manager is enabled. As a result, `@throws SecurityException` will appear on fewer methods in the API specification.

Deprecate APIs for removal

The Security Manager consists of the class `java.lang.SecurityManager` and a number of closely related APIs in the `java.lang` and `java.security` packages. We will terminally deprecate the following eight classes and two methods, by annotating them with `@Deprecated(forRemoval=true)`:

- **`java.lang.SecurityManager`** — The primary API for the Security Manager.
- **`java.lang.System::{setSecurityManager, getSecurityManager}`** — Methods for setting and getting the Security Manager.
- **`java.security.{Policy, PolicySpi, Policy.Parameters}`** — The primary APIs for policy, which are used to determine if code running under the Security Manager has been granted permission to perform specific privileged operations.
- **`java.security.{AccessController, AccessControlContext, AccessControlException, DomainCombiner}`** — The primary APIs for the access controller, which is the default implementation to which the Security Manager delegates permission checks. These APIs do not have value without the Security Manager, since certain operations will not work without both a policy implementation and access-control context support in the VM.

We will also terminally deprecate the following two classes and eight methods which depend strongly on the Security Manager:

- **`java.lang.Thread::checkAccess`**, **`java.lang.ThreadGroup::checkAccess`**, and **`java.util.logging.LogManager::checkAccess`** — These three methods are anomalous because they allow ordinary Java code to check whether it would be trusted to perform certain operations without actually performing them. They serve no purpose without the Security Manager.
- **`java.util.concurrent.Executors::{privilegedCallable, privilegedCallableUsingCurrentClassLoader, privilegedThreadFactory}`** — These utility methods are only useful when the Security Manager is enabled.
- **`java.rmi.RMISecurityManager`** — RMI's Security Manager class. This class is obsolete and was deprecated in Java 8.
- **`javax.security.auth.SubjectDomainCombiner`** and **`javax.security.auth.Subject::{doAsPrivileged, getSubject}`** — APIs for user-based authorization that are dependent on Security Manager APIs such as `AccessControlContext` and `DomainCombiner`. We plan to provide a replacement API for `Subject::getSubject` since it is commonly used for use cases that do not require the Security Manager, and to continue to support use cases involving `Subject::doAs` (see below).

We will not deprecate some classes in the `java.security` package that are related to the Security Manager, for various reasons:

- **`SecureClassLoader`** — The superclass of `java.net.URLClassLoader`. Also, as of Java 9, `SecureClassLoader` is instrumental in the implementation of the application class loader and the platform class loader.
- **`CodeSource`** — Although most often associated with granting permissions based on the location of code, `CodeSource` is not directly tied to the Security Manager and can provide independent value as a means to identify the source of a body of code and, optionally, who signed it.
- **`ProtectionDomain`** — Several significant APIs depend on `ProtectionDomain`, e.g., `ClassLoader::defineClass` and `Class::getProtectionDomain`. `ProtectionDomain` also has value

independent of the Security Manager since it contains the `CodeSource` of a class.

- **`Permission`** and subclasses — Other significant classes, such as `ProtectionDomain`, depend on `Permission`. Many of the subclasses of `Permission`, however, are specific to use cases which will likely no longer be relevant after the Security Manager is removed. The maintainers of these subclasses can deprecate and remove them separately, after evaluating the compatibility risk.
- **`PermissionCollection`** and **`Permissions`** — Classes that hold collections of `Permission` objects and do not depend directly on the Security Manager.
- **`PrivilegedAction`**, **`PrivilegedExceptionAction`**, and **`PrivilegedActionException`** — These APIs do not depend directly on the Security Manager and are used by the `javax.security.auth` API for authentication and authorization (see below).
- **`SecurityException`** — A runtime exception thrown by Java APIs when a permission check fails. We may deprecate this API for removal at a later date, but for now the impact of doing so would be too high.

We will not deprecate the `javax.security.auth.Subject::doAs` method since it can be used to transport a `Subject` across API boundaries by attaching it to the thread's `AccessControlContext`, serving a purpose similar to a `ThreadLocal`. The credentials of the `Subject` can then be obtained by an underlying authentication mechanism (e.g., a Kerberos implementation of GSSAPI) by calling `Subject::getSubject`. These credentials can be used for authentication or authorization purposes and do not require the Security Manager to be enabled. However, `Subject::doAs` depends on APIs tightly related to the Security Manager, such as `AccessControlContext` and `DomainCombiner`. Thus, we plan to create a new API that does not depend on the Security Manager APIs; subsequently we will then deprecate the `Subject::doAs` API for removal.

We will not deprecate any tools. (We removed the `policytool` GUI for editing policy files in JDK 10.)

Issue warnings

We will make the following changes to ensure that developers and users are aware that the Security Manager is deprecated for removal.

- If the default Security Manager or a custom Security Manager is enabled at startup:

  ```
  java -Djava.security.manager                    MyApp
  java -Djava.security.manager=""                 MyApp
  java -Djava.security.manager=default            MyApp
  java -Djava.security.manager=com.foo.bar.Server MyApp
  ```

  then the following warning is issued at startup:

  ```
  WARNING: A command line option has enabled the Security Manager
  WARNING: The Security Manager is deprecated and will be removed in a future release
  ```

  This warning, unlike compile-time deprecation warnings, cannot be suppressed.

  (The four invocations of `java -D...` shown above set the system property `java.security.manager` to, respectively, the empty string, the empty string, the string `default`, and the class name of a custom Security Manager. These invocations were the supported ways of enabling a Security Manager at startup in Java releases prior to Java 12. Java 12 added support for the strings `allow` and `disallow`, shown next.)

- If a Security Manager is not enabled at startup, but could be installed dynamically during run time:

  ```
  java MyApp
  java -Djava.security.manager=allow MyApp
  ```

  then no warning is issued at startup. Instead, a warning is issued at run time when `System::setSecurityManager` is called, as follows:

  ```
  WARNING: A terminally deprecated method in java.lang.System has been called
  WARNING: System::setSecurityManager has been called by com.foo.bar.Server (file:/tmp/foobarserver/thing.jar)
  WARNING: Please consider reporting this to the maintainers of com.foo.bar.Server
  WARNING: System::setSecurityManager will be removed in a future release
  ```

  This warning is shown once per caller, and unlike compile-time deprecation warnings, cannot be suppressed.

- If a Security Manager is not enabled at startup, and the system property `java.security.manager` is set to `disallow`:

  ```
  java -Djava.security.manager=disallow MyApp
  ```

  then no warning is issued at startup, nor at run time if an attempt is made to install a Security Manager dynamically by invoking `System::setSecurityManager`. However, every invocation of `System::setSecurityManager` throws an `UnsupportedOperationException` with the following detail message:

  ```
  The Security Manager is deprecated and will be removed in a future release
  ```

  In Java 18, `disallow` will become the default value of `java.security.manager`. The command line `java MyApp` will then have the same effect as `java -Djava.security.manager=disallow MyApp` had on Java 17.

Future Work

This JEP is about deprecating the Security Manager for removal in the future; it does not propose to remove the Security Manager now. Thus there is time to consider use cases where the Security Manager is useful today and where

developing replacements of, or alternatives to, some of its functionality may be justified. Here is a list of potential enhancements and current work in progress:

- **Securing access to native code** — Applications that run with the Security Manager can use permissions to prevent native code from being loaded, so that it cannot be accessed via the Java Native Interface (JNI). The planned replacement for JNI is the Foreign Function & Memory API (JEP 412), which provides a Java API for interoperating with code and data outside the Java runtime; it will protect access to native code without depending on the Security Manager.
- **Monitoring access to resources** — Applications that run with the Security Manager sometimes use it to monitor or log operations such as file and network access, while not necessarily restricting these operations. There are potentially better ways to monitor these types of activities, such as using JDK Flight Recorder. We will evaluate adding new JFR events for networking, filesystem, and process creation with the aim of increasing application security and providing insights into the platform APIs that perform these kinds of operations.
- **Blocking `System::exit`** — Some IDEs and frameworks use a custom Security Manager to prevent applications from calling this method. This use case may benefit from a new API.
- **Securing deserialization** — Applications that run with the Security Manager and deserialize data are vulnerable to attack if permissions are not granted correctly (see, e.g., Java Secure Coding Guideline 8-5). Alternatively, Serialization Filters (JEP 290) allow incoming data to be validated early, and Context-Specific Deserialization Filters (JEP 415) will improve the flexibility and granularity of that validation.
- **Securing XML processing** — As described in the Motivation, JAXP has a mode that processes XML more securely. This mode is opt-in but it is also enabled by default when applications run with the Security Manager. We will investigate enabling this mode by default, regardless of whether a Security Manager is enabled. (XML Signature has a similar secure validation mode that is opt-in but enabled by default when the Security Manager is enabled. As of Java 17 this mode is enabled by default, regardless of whether a Security Manager is enabled.)

Alternatives

- **Preserve the Security Manager API for extension by custom Security Managers that wish to intercept, log, and veto access to resources** — Remove support for policy files, but retain the permission checking mechanics that are woven into the Java class libraries.

  This option forces developers to learn the principles and best practices of the Security Manager architecture, including the complex science of permission checking, to achieve the much simpler goal of resource monitoring. It would also likely raise questions about whether it would be worth keeping all of the permission-checking, Security Manager-calling hooks in the JDK — at `System::exit`, yes, at `System::getProperty`, maybe not. We think that we should instead investigate improving options that provide similar features, such as JDK Flight Recorder.
- **Enhance the Security Manager** — Enhancing the Security Manager to address new use cases, or to fix its many drawbacks, is not practical. Enabling the Security Manager by default, running code with `AllPermission`, and logging all the permission checks as a way to encourage developers to take it more seriously is not sensible.
- **Leave the Security Manager in place** — Continue to support it as-is, without investing further to improve it.

Each of these alternatives requires retaining the Security Manager in something close to its current form. After decades of maintaining the Security Manager but seeing very little usage, we are no longer willing to bear this ongoing and expensive burden.

Testing

We will add new tests to verify that warnings are issued when the Security Manager is enabled on the command line or dynamically installed at run time.

Risks and Assumptions

- The Security Manager has been part of the Java Platform since JDK 1.0, so some applications may be impacted by its deprecation and eventual removal. The full functionality of the Security Manager will, however, be maintained in the release to which this JEP is targeted. Applications can continue to rely on supported JDKs for some time as they migrate to newer APIs and mechanisms.
- Jakarta EE has several requirements on the Security Manager. We assume that these requirements will be relaxed or removed in order for compliant applications to run on future Java releases after the Security Manager is degraded and then removed.