JEP 451: Prepare to Disallow the Dynamic Loading of Agents

Author	Ron Pressler & Alex Buckley
Owner	Ron Pressler
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	hotspot / svc
Discussion	jigsaw dash dev at openjdk dot org, serviceability dash dev at openjdk dot org
Reviewed by	Alan Bateman, Dan Heidinga
Endorsed by	Mark Reinhold, Serguei Spitsyn
Created	2023/04/18 09:39
Updated	2023/08/21 14:44
Issue	8306275

Summary

Issue warnings when agents are loaded dynamically into a running JVM. These warnings aim to prepare users for a future release which disallows the dynamic loading of agents by default in order to improve integrity by default. Serviceability tools that load agents at startup will not cause warnings to be issued in any release.

Goals

- Prepare for a future release of the JDK that will, by default, disallow the loading of agents into a running JVM.
- Reassess the balance between serviceability, which involves ad-hoc changes to running code, and integrity, which assumes that running code is not arbitrarily changed.
- Ensure that the majority of tools — which do not need to load agents dynamically — are unaffected.
- Align the ability to load agents dynamically with other so-called "superpower" capabilities, such as deep reflection.

Non-Goals

- It is not a goal to prevent agents from being loaded at JVM startup via the `-javaagent` or `-agentlib` command-line options, nor to issue warnings upon such use.
- It is not a goal to deprecate or remove the parts of the Attach API which load agents dynamically; it is only a goal to prepare for their use being disallowed by default.
- It is not a goal to change the parts of the Attach API which allow serviceability tools to connect to a running JVM for monitoring and management purposes. Tools such as `jcmd` and `jconsole` will continue to work without command line options and without warnings.

Motivation

Agents in the Java Platform

An *agent* is a component that can alter the code of an application while the application is running. Agents were introduced by the Java Platform Profiling Architecture in JDK 5 as a way for tools, notably profilers, to *instrument* classes. This means altering the code in a class so that it emits events to be consumed by a tool outside the application, without otherwise changing the code's behavior. Agents achieve this either by transforming classes during class loading, or by redefining classes loaded earlier. They can be written in Java code using the `java.lang.instrument` API ("Java agents"), or in native code using the JVM Tool Interface ("JVM TI agents").

Agents were designed with benign instrumentation in mind, where the addition of instrumentation does not affect application behavior. However, advanced developers found use cases such as Aspect-Oriented Programming that change application behavior in arbitrary ways. There is also nothing to stop an agent from altering code outside the application, such as code in the JDK itself. To ensure that the owner of an application approved the use of agents, JDK 5 required agents to be specified on the command line with the `-javaagent` or `-agentlib` options, and loaded the agents immediately at startup. This represented an explicit grant of privileges by the application owner.

Serviceability and dynamically loaded agents

Serviceability is the ability of a system operator to monitor, observe, debug and troubleshoot an application while it runs. The Java Platform's excellent serviceability has long been a source of pride.

To support serviceability tools, JDK 6 introduced the Attach API. The Attach API is not part of the Java Platform but, rather, a JDK API supported for external use. It allows a tool launched with appropriate operating-system privileges to connect to a running JVM, either local or remote, and communicate with that JVM to observe and control its operation. The Attach API is enabled by default but can be disabled with the `-XX:+DisableAttachMechanism` option on the command line.

Examples of tools that use the Attach API include:

- Monitoring and management tools, such as `jcmd` and `jconsole`, which observe application metrics and alter configurations. For example, if an application uses the `java.util.logging` API then an operator can use `jconsole` to dynamically alter the log level. These tools make use of specialized `jcmd` protocols, JMX, and the JDK Flight Recorder (JFR).

- Debuggers, which require an agent built into the JVM to be enabled at startup with the `-agentlib:jdwp` option. They then communicate with the agent over some IPC channel, but are also able to take advantage of the Attach API.
- Profilers, and more generally Application Performance Monitoring (APM) tools, which use an agent loaded at startup to instrument application code so that it emits JFR events for consumption by JDK Mission Control or other clients.

The Attach API also allows a tool to load an agent dynamically, into a running JVM. This capability supports advanced use cases that involve altering arbitrary code on the fly. Examples of tools that load agents dynamically include:

- Profilers, which connect to a running JVM and load an agent dynamically to instrument application code.
- Ad-hoc troubleshooting tools, which read and write application state at run time. The dynamically loaded agent either uses JVM TI to inspect the running program's state, or transforms and instruments loaded classes.

(Very advanced developers sometimes fix bugs in production environments by writing an agent that patches buggy code and loading that agent dynamically. However, this is not a supported use case and has never been recommended. An agent's ability to redefine loaded classes is subject to limitations, so the ability to fix bugs by patching is limited. Moreover, an agent cannot persist the changes that it makes, so restarting the application will revert the change.)

Dynamically loaded agents give serviceability tools the superpower to alter a running application. However, attaching a tool is triggered by a human operator with appropriate operating-system credentials. This human in the loop grants approval to alter the application, so serviceability tools are not subject to the integrity constraints imposed upon other code. Therefore the dynamic loading of agents has been allowed by default, though in JDK 9 and above it can be disallowed with the `-XX:-EnableDynamicAgentLoading` option on the command line.

Agents and libraries

Despite a conceptual separation of concerns between libraries and tools, some libraries provide functionality that relies upon the code-altering superpower afforded to agents. For example, a mocking library might redefine application classes to bypass business-logic invariants, while a white-box testing library might redefine JDK classes so that reflection over `private` fields is always permitted. To obtain these capabilities, a library can employ an agent that obtains an all-powerful `Instrumentation` object from the JVM and conveys it to the library.

Some such libraries ensure that the application owner grants approval to alter the application by requiring the library's agent to be specified on the command line with the `-javaagent` option. An example of a library that did this was Quasar, an early prototype of what later became Virtual Threads (JEP 444).

Other libraries take a more dubious approach, obtaining capabilities without the approval of the application owner. They use the Attach API to silently connect to the JVMs in which they run and load agents dynamically, in effect masquerading as serviceability tools. To maintain integrity, JDK 9 and later releases prevent code from connecting to the current JVM by default. (Such connections can be enabled via `-Djdk.attach.allowAttachSelf=true`.) However, that measure has proven insufficient: Some libraries now spawn a second JVM which connects to the first and loads the agent there, alongside the library.

If a library uses agents to silently redefine JDK classes, thereby bypassing strong encapsulation, then none of the invariants enforced by strong encapsulation can be trusted. Integrity is lost.

Toward integrity by default

To assure integrity, we need stronger measures to prevent the misuse by libraries of dynamically loaded agents. Unfortunately, we have not found a simple and automatic way to distinguish between a serviceability tool that dynamically loads an agent and a library that dynamically loads an agent. Giving free reign to tools would imply giving free reign to libraries, which is tantamount to giving up on integrity by default.

We therefore propose to require the dynamic loading of agents to be approved by the application owner — just as we have required the startup-time loading of agents to be approved by the application owner since JDK 5. This change will move the Java Platform closer to the long-term vision of integrity by default. In practical terms, the application owner will have to choose to allow the dynamic loading of agents via a command line option.

Fortunately, most serviceability tools do not rely upon dynamically loaded agents. However, disallowing the dynamic loading of agents by default means that ad-hoc troubleshooting techniques that require dynamically loading an agent will no longer work out-of-the-box. If the need arises to dynamically load an agent then the JVM must be restarted with the appropriate command-line option in order to grant the application owner's approval.

The impact of this change will be mitigated by the fact that most modern server applications are designed with redundancy, so individual nodes can be restarted with the command line option as needed. Special cases — such as a JVM that must never be stopped for maintenance, or a canary process for a new software version which is kept under close observation — can typically be identified in advance so that the dynamic loading of agents can be enabled from the start.

Requiring application owners to grant approval for dynamically loaded agents will allow the Java ecosystem to attain the vision of integrity by default without substantially constraining serviceability.

Description

In JDK 21, the dynamic loading of agents is allowed but the JVM issues a warning when it occurs. For example:

```
WARNING: A {Java,JVM TI} agent has been loaded dynamically (file:/u/bob/agent.jar)
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
```

To allow tools to dynamically load agents without warnings, users must run with the `-XX:+EnableDynamicAgentLoading` option on the command line.

Running with `-Djdk.instrument.traceUsage` causes the methods of the `java.lang.instrument` API to print a message and a stack trace when used. This helps identify libraries that incorrectly use dynamically loaded agents instead of agents loaded at startup. Maintainers of libraries that load agents dynamically are encouraged to update their documentation to describe how users can load agents at startup; the various deployment options are given by the java.lang.instrument API.

In some future release, the dynamic loading of agents will be disallowed by default. Out of the box, any use of the Attach API to dynamically load an agent will cause an exception to be thrown:

```
com.sun.tools.attach.AgentLoadException: Failed to load agent library: \
Dynamic agent loading is not enabled. Use -XX:+EnableDynamicAgentLoading \
to launch target VM.
```

To allow the dynamic loading of agents when it is disallowed by default, users must run with `-XX:+EnableDynamicAgentLoading` on the command line.

To prepare for the changed default in the future release, users of JDK 9 or any later release can explicitly disallow the dynamic loading of agents by running with `-XX:-EnableDynamicAgentLoading` on the command line.

Tools that employ agents loaded at startup are unaffected by these changes. The meaning and operation of the `-javaagent` option, the `-agentlib` option, and the `Launcher-Agent-Class` JAR-file attribute are unchanged.

Tools that use the Attach API for purposes other than dynamically loading an agent are unaffected by these changes.

Libraries must not load agents dynamically. Libraries employing an agent must load it at startup with the `-javaagent`/`-agentlib` options.

Historical note

Disallowing the dynamic loading of agents by default was originally proposed in 2017 as part of adding modules to the platform in JDK 9. The proposal was:

> The dynamic loading of JVM TI agents will be disabled by default in a future release. To prepare for that change we recommend that applications that allow dynamic agents start using the option `-XX:+EnableDynamicAgentLoading` to enable that loading explicitly.

The consensus in 2017 was to defer the change from JDK 9 to a later release so that tool maintainers would have time to inform their users. Nevertheless, when we have strengthened encapsulation in the past we have issued warnings in a preceding release in order to build awareness of the upcoming change. This JEP follows the same procedure.

Risks and Assumptions

- We assume that most serviceability scenarios involve the use of `jcmd`, `jconsole`, debuggers, JFR, and APM tools, which do not dynamically load agents and will therefore not be affected.
- We assume that maintainers of libraries which dynamically load an agent will update their documentation to ask application owners to load the agent at startup with the `-javaagent` option, or else enable the dynamic loading of agents via the `-XX:+EnableDynamicAgentLoading` option.

Future Work

- Advanced profilers that profile native code use JVM TI agents merely to gain access to internal HotSpot mechanisms that can support profiling. When profiling an application in production, they may load the agent dynamically. This use case is best addressed by expanding the capabilities of JFR to perform the task without requiring an agent at all. JFR is able to cooperate with HotSpot's JIT compiler to capture large batches of stack traces far more efficiently than anything that could be exposed through the JVM TI API or the internal, undocumented, `AsyncGetCallTrace` method that is commonly used by advanced profilers.
- Some interesting use cases for code manipulation could be served by offering an encapsulation-respecting `Instrumentation` object to libraries directly, with no agent involved. This would allow a library to transform or redefine classes in modules that are open to the library's module.

Alternatives

- Warn only on the dynamic loading of native JVM TI agents by default, and restrict the capabilities of dynamically loaded Java agents by default (i.e., when the `-XX:+EnableDynamicAgentLoading` option is not specified) such that a warning is issued when they attempt to modify classes in named modules while allowing them to modify classes in unnamed modules without warning.

  This approach is more complex and does not support substantially more practical tooling agents. Moreover, it does not preclude a Java agent from employing JNI to grant itself further powers.
- Employ an authentication mechanism that distinguishes between a human-operated tool and a library masquerading as a tool to, by default, allow a tool to dynamically load an agent without warning but warn when a library attempts to dynamically load an agent.

  We have explored several approaches along these lines, but all were either complex or required special setup on the command line, which would not

reduce the impact on tools that dynamically load an agent.