ENV05-J. Do not deploy an application that can be remotely monitored

Java provides several APIs that allow external programs to monitor a running Java program. These APIs also permit the Java program to be monitored remotely by programs on distinct hosts. Such features are convenient for debugging the program or fine-tuning its performance. However, if a Java program is deployed in production with remote monitoring enabled, an attacker can connect to the Java Virtual Machine (JVM) and inspect its behavior and data, including potentially sensitive information. An attacker can also exert control over the program's behavior. Consequently, remote monitoring must be disabled when running a Java program in production.

JVM Tool Interface (JVMTI)

Java 5 introduced the JVM Tool Interface (JVMTI) [Sun 04d], replacing both the JVM Profiler Interface (JVMPI) and the JVM Debug Interface (JVMDI), which are now deprecated.

The JVMTI contains extensive facilities to learn about the internals of a running JVM, including facilities to monitor and modify a running Java program. These facilities are rather low level and require the use of the Java Native Interface (JNI) and C language programming. However, they provide the opportunity to access fields that would normally be inaccessible. Also, there are facilities that can change the behavior of a running Java program (for example, threads can be suspended or stopped). The JVMTI profiling tools can also measure the time that a thread takes to execute, leaving applications vulnerable to timing attacks.

The JVMTI works by using agents that communicate with the running JVM. These agents must be loaded at JVM startup and are usually specified via one of the command-line options `-agentlib` or `-agentpath`. However, agents can be specified in environment variables, although this feature can be disabled where security is a concern. The JVMTI is always enabled, and JVMTI agents may run under the default security manager without requiring any permissions to be granted.

Java Platform Debugger Architecture (JPDA)

The Java Platform Debugger Architecture (JPDA) builds on the JVMTI and provides high-level facilities for debugging Java systems while they are running [JPDA 2004].

The JPDA facilities are similar to the reflection API, which is described in SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields. In particular, the JPDA provides methods to get and set field and array values. Access control is not enforced, so even the values of private fields can be set by a remote process via the JPDA.

Various permissions must be granted for debugging to take place under the default security manager. The following policy file was used to run the JPDA Trace demonstration under the default security manager:

```
grant {
  permission java.io.FilePermission "traceoutput.txt", "read,write";
  permission java.io.FilePermission "C:/Program Files/Java/jdk1.5.0_04/lib/tools.jar", "read";
  permission java.io.FilePermission "C:/Program", "read,execute";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
  permission java.lang.RuntimePermission "accessClassInPackage.sun.misc";
  permission java.lang.RuntimePermission "loadLibrary.dt_shmem";
  permission java.util.PropertyPermission "java.home", "read";
  permission java.net.SocketPermission "<localhost>", "resolve";
  permission com.sun.jdi.JDIPermission "virtualMachineManager";
};
```

Because JPDA supports remote debugging, a remote host can access the debugger. An attacker can exploit this feature to study sensitive information or modify the behavior of a running Java application unless appropriate protection is enabled. A security manager can ensure that only known, trusted hosts are given permissions to use the debugger interface.

Java SE Monitoring and Management Features

Java contains extensive facilities for monitoring and managing a JVM [JMX 2006]. In particular, the Java Management Extension (JMX) API enables the monitoring and control of class loading, thread state and stack traces, deadlock detection, memory usage, garbage collection, operating system information, and other operations [Sun 04a]. It also has facilities for logging monitoring and management.

The Java SE monitoring and management features fall into four broad categories:

- **The JMX technology:** This technology serves as the underlying interface for local and remote monitoring and management.
- **Instrumentation for the JVM:** These facilities enable out-of-the-box monitoring and management of the JVM and are based on the JMX specification.
- **Monitoring and management API:** These facilities use the `java.lang.management` package to provide the monitoring and management interface. Applications can use this package to monitor themselves or to let JMX technology–compliant tools monitor and manage them.
- **Monitoring and management tools:** Tools such as JConsole implement the JMX interface to provide monitoring and management facilities.

These facilities can be used either locally (on the machine that runs the JVM) or remotely. Local monitoring and management is enabled by default when a JVM is started; remote monitoring and management is not. For a JVM to be monitored and managed remotely, it must be started with various system properties set (either on the command line or in a configuration file).

When remote monitoring and management is enabled, access is password-controlled by default. However, password control can be disabled. Disabling password authentication is insecure because any user who can discover the port number that the JMX service is listening on can monitor and control the Java applications running on the JVM [JMXG 2006].

The JVM remote monitoring and management facility uses a secure communication channel (Secure Sockets Layer [SSL]) by default. However, if an attacker can start a bogus remote method invocation (RMI) registry server on the monitored machine before the legitimate RMI registry server is started, JMX passwords can be intercepted. Also, SSL can be disabled when using remote monitoring and management, which could, again, compromise security. See *The Java SE Monitoring and Management Guide* [JMXG 2006] for further details and for mitigation strategies.

There are also provisions to require proper authentication of the remote server. However, users may start a JVM with remote monitoring and management enabled but with no security; this would leave the JVM open to attack by outsiders. Although accidentally enabling remote monitoring and management is unlikely, users might not realize that starting a JVM so enabled, without any security, could leave their JVM exposed to attack.

If exploited, the monitoring and management facilities can seriously compromise the security of Java applications. For example, an attacker can obtain information about the number of classes loaded and threads running, thread state along with traces of live threads, system properties, virtual machine arguments, and memory consumption.

Noncompliant Code Example (JVMTI)

In this noncompliant code example, the JVMTI works by using agents that communicate with the running JVM. These agents are usually loaded at JVM startup via one of the command-line options `-agentlib` or `-agentpath`. In the following command, `libname` is the name of the library to load while `options` are passed to the agent on startup.

```
${JDK_PATH}/bin/java -agentlib:libname=options ApplicationName
```

Some JVMs allow agents to be started when the JVM is already running. This practice is insecure in a production environment. Refer to the JVMTI documentation [JVMTI 2006] for platform-specific information on enabling/disabling this feature.

Platforms that support environment variables allow agents to be specified in such variables. "Platforms may disable this feature in cases where security is a concern; for example, the Reference Implementation disables this feature on UNIX systems when the effective user or group ID differs from the real ID" [JVMTI 2006].

Agents may run under the default security manager without requiring any permissions to be granted. Although the JVMTI is useful for debuggers and profilers, such levels of access are inappropriate for deployed production code.

Noncompliant Code Example (JPDA)

This noncompliant code example uses command-line arguments to invoke the JVM so that it can be debugged from a running debugger application by listening for connections using shared memory at transport address `mysharedmemory`:

```
${JDK_PATH}/bin/java -agentlib:jdwp=transport=dt_shmem,
    address=mysharedmemory ApplicationName
```

Likewise, the command-line arguments `-Xrunjdwp` (which is equivalent to `-agentlib`) and `-Xdebug` (which is used by the `jdb` tool) also enable application debugging.

Noncompliant Code Example (JVM monitoring)

This noncompliant code example invokes the JVM with command-line arguments that permit remote monitoring via port 8000. This invocation may result in a security vulnerability when the password is weak or the SSL protocol is misapplied.

```
${JDK_PATH}/bin/java
    -Dcom.sun.management.jmxremote.port=8000 ApplicationName
```

Compliant Solution

This compliant solution starts the JVM without any agents enabled. Avoid using the `-agentlib`, `-Xrunjdwp`, and `-Xdebug` command-line arguments on production machines. This compliant solution also installs the default security manager.

```
${JDK_PATH}/bin/java -Djava.security.manager ApplicationName
```

Clear the environment variable `JAVA_TOOL_OPTIONS` in the manner appropriate for your platform, for example, by setting it to an empty string value. Doing prevents JVMTI agents from receiving arguments via this mechanism. The command-line argument `-Xnoagent` can also be used to disable the debugging features supported by the old Java debugger (`oldjdb`).

This compliant solution disables monitoring by remote machines. By default, local monitoring is enabled in Java 6 and later. In earlier versions, the system property `com.sun.management.jmxremote` must be set to enable local monitoring. Although the unsupported `-XX:+DisableAttachMechanism` com mand-line option may be used to disable local Java tools from monitoring the JVM, it is always possible to use native debuggers and other tools to perform monitoring. Fortunately, monitoring tools require at least as many privileges as the owner of the JVM process possesses, reducing the threat of a local privilege escalation attack.

Local monitoring uses temporary files and sets the file permissions to those of the owner of the JVM process. Ensure that adequate file protection is in place on the system running the JVM so that the temporary files are accessed appropriately (see FIO03-J. Remove temporary files before termination for additional information).

The *Java SE Monitoring and Management Guide* [JMXG 2006] provides further advice:

> *Local monitoring with `jconsole` is useful for development and prototyping. Using `jconsole` locally is not recommended for production environments because `jconsole` itself consumes significant system resources. Rather, use `jconsole` on a remote system to isolate it from the platform being monitored.*

Moving `jconsole` to a remote system removes its system resource load from the production environment.

Noncompliant Code Example (Remote Debugging)

Remote debugging requires the use of sockets as the transport (`transport=dt_socket`). Remote debugging also requires specification of the type of application (`server=y`, where `y` denotes that the JVM is the server and is waiting for a debugger application to connect to it) and the port number to listen on (`address=9000`).

```
${JDK_PATH}/bin/java -agentlib:jdwp=transport=dt_socket,
    server=y,address=9000 ApplicationName
```

Remote debugging is dangerous because an attacker can spoof the client IP address and connect to the JPDA host. Depending on the attacker's position in the network, he or she could extract debugging information by sniffing the network traffic that the JPDA host sends to the forged IP address.

Compliant Solution (Remote Debugging)

Restrict remote debugging to trusted hosts by modifying the security policy file to grant appropriate permissions only to those trusted hosts. For example, specify the permission `java.net.SocketPermission` for only the JPDA host and remove the permission from other hosts.

The JPDA host can serve either as a server or as a client. When the attacker cannot sniff the network to determine the identity of machines that use the JPDA host (for example, through the use of a secure channel), specify the JPDA host as the client and the debugger application as the server by changing the value of the `server` argument to `n`.

This compliant solution allows the JPDA host to attach to a trusted debugger application:

```
${JDK_PATH}/bin/java -agentlib:jdwp=transport=dt_socket,
    server=n,address=9000 ApplicationName
```

When it is necessary to run a JVM with debugging enabled, avoid granting permissions that are not needed by the application. In particular, avoid granting socket permissions to arbitrary hosts; that is, omit the permission `java.net.SocketPermission "*", "connect,accept"`.

Exceptions

ENV05-J-EX0: A Java program may be remotely monitored using any of these technologies if it can be guaranteed that no program outside the local trust boundary can access the program. For example, if the program lives on a local network that is both completely trusted and disconnected from any untrusted networks, including the Internet, remote monitoring is permitted.

Risk Assessment

Deploying a Java application with the JVMTI, JPDA, or remote monitoring enabled can allow an attacker to monitor or modify its behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV05-J	High	Probable	Low	P18	L1

Automated Detection

This rule is not amenable to automated static analysis.

Related Vulnerabilities

CVE-2010-4495 describes a vulnerability in the TIBCO ActiveMatrix product line in which a flaw in JMX connection processing allowed remote users to execute arbitrary code, cause denial of service, or obtain potentially sensitive information.

Android Implementation Details

JVMTI is not supported on the Dalvik VM.

Bibliography

[JMX 2006]	
[JMXG 2006]	
[JPDA 2004]	
[JVMTI 2006]	
[Long 2005]	Section 2.6, "The JVM Tool Interface" Section 2.7, "Debugging" Section 2.8, "Monitoring and Management"
[Reflect 2006]	Reflection