



## Problem determination for javacore files from WebSphere Application Server

### Technote (FAQ)

#### Problem

Some error conditions in an IBM® Java™ virtual machine (JVM™) running under WebSphere® Application Server result in a crash. The output of a crash is a javacore file. This document describes how to read and interpret the data in a javacore file.

Performing problem determination by analyzing the javacore file is an effective means of determining root cause; taking preventative action helps to avoid future crashes of the JVM.

These problem determination techniques are applicable to the Java Software Developer Kit (Java SDK) delivered by IBM® on Microsoft® Windows®, Linux® and AIX®.

#### Cause

A javacore file is a snapshot of a running Java process. The IBM Java SDK produces a javacore in response to specific operating system signals. Javacore files show the state of every thread in the Java process, as well as the monitor information identifying Java synchronization locks.

You can use Javacore files to resolve the problems listed below. For more information on capturing data for problem determination, see the MustGather documents for these problems:

- [100% CPU Usage](#)
- [Crash](#)
- [Hang/Performance Degradation](#)

#### Notes:

- Application Server restarts often indicate a JVM crash.
- Sometimes a JVM crash does not produce a javacore, in which case it might be necessary to disable the javacore processing to allow a core file to be created on UNIX® or `user.dmp` file to be created on Microsoft Windows.

#### Solution

Table of contents for problem determination:

---

<b><u>1. Locating and identifying javacore files</u></b>
<u>1.1 Locating where a javacore file is written is determined by the following</u>
<u>1.2 Javacore naming</u>
<b><u>2. How javacore files (thread dumps) are created</u></b>
<u>2.1 Operating system signals</u>
<u>2.1.1 UNIX signals</u>
<u>2.1.2 Windows signals</u>
<u>2.2 Thread dump via an explicit action by the customer</u>
<b><u>3. Reading javacore files</u></b>
<u>3.1 Signal information</u>
<u>3.1.1 Time and date of the javacore</u>
<u>3.1.2 Type of signal and location of the JVM process at the time of the javacore</u>
<u>3.1.3 Java fullversion</u>
<u>3.1.4 Current thread details</u>
<u>3.2 Operating environment</u>
<u>3.2.1 Signal handlers</u>
<u>3.2.2 Loaded libraries</u>
<u>3.2.3 Java command line</u>
<u>3.2.3.1 Determining which WebSphere Application</u>

Server process a javacore is for3.3 JVM monitor information3.4 The JVM thread dump4. How to proceed if the library that caused the crash is not identified5. What can prevent a javacore file from being produced?**1. Locating and identifying javacore files****1.1. Locating where a javacore file is written is determined by the following**

1. The setting of the IBM\_JAVACOREDIRENvironment variable.
2. If this variable is not set, the javacore is written to the working directory of the JVM; use the `-DWORKING_DIR` parameter on the JVM command line to set this variable.
3. Otherwise, the javacore is written to the directory from which the Java process was started.

- **V4.0 release Java processes**

For WebSphere Application Server Java processes, this is usually the `install_root/bin` directory.

- **V5.0 release Base Edition Java processes**

For WebSphere Application Server Java processes, this is usually the `install_root/bin` directory.

[back to top](#) ↑

**1.2 Javacore naming**

The name of the javacore file (for example, `javacore24802.1026159146.txt`) is derived from a couple of system values. Each javacore has a unique name. The following table indicates the name format for each operating system.

Operating System	Javacore file name format	Meaning
Windows and Linux	javacore.YYYYMMDD.HHMMSS.PID.txt	YYYY=year, MM=month, DD=day, SS=second, PID=processID
AIX	javacorePID.TIME.txt	PID=processID, TIME=time since 1/1/1970
z/OS	Javadump.YYYYMMDD.HHMMSS.PID.txt	YYYY=year, MM=month, DD=day, SS=second, PID=processID

The *PID* is the process ID of the JVM that dumped the javacore. The PID can be used in conjunction with the WebSphere Application Server logs to determine which WebSphere Application Server process the javacore is for. For Linux, each thread is its own process, so any JVM appears to have several running processes on Linux. The PID is the one with which the JVM started.

[back to top](#) ↑

**2. How javacore files (thread dumps) are created**

The IBM Java SDKs produce a javacore file in response to an operating system signal. The JVM has signal handlers that handle operating system signals. An operating system signal can be raised because of a run-time exception, or by an explicit action by the user.

[back to top](#) ↑

**2.1 Operating system signals**

Operating system signals are raised in native code. Java itself will not issue a signal to end the JVM process. An operating system signal that causes the JVM process to end is caused by native code that the JVM calls. Native code libraries are used for a variety of different functions; for example, database calls.

[back to top](#) ↑

**2.1.1 UNIX Signals**

The javacore displays the operating system signal that caused the javacore to be written. Some signals also produce a core file in UNIX operating systems. The

JVM signal handler library is `libhpi.a`.

Following are some of the most commonly seen signals in javacore files:

- **SIGQUIT**  
This signal is sent when a **kill -3** command is issued against the JVM process. This signal typically does not end the JVM process. It generates a thread dump (javacore) for diagnosing a potentially hung JVM process.
- **SIGILL**  
This is the equivalent of a **kill -4** command. This means an illegal instruction was executed. This can mean a corruption of the code segment or an branch that is not valid within the native code. This signal often indicates a problem caused by JIT-compiled code.
- **SIGSEGV**  
This signal is equivalent to a **kill -11** command. It indicates an operation that is not valid in a program, such as accessing an illegal memory address. This is typically indicative of a programming problem in one of the native libraries.

For more information on signals in WebSphere Application Server, refer to the technote [What is a signal and why does this matter for WebSphere Application Server](#).

[back to top](#) ↑

### 2.1.2 Windows Signals

The signals that Windows uses are entirely different than those used by UNIX. Here is a list of some of the operating system signals produced on Windows. Some signals also produce a user.dmp file. The JVM signal handler library is `hpi.dll`.

Signal Name	Description	JVM action
Memory access error	Invalid memory address	javacore and abort
Illegal access error		javacore and abort

[back to top](#) ↑

## 2.2 Thread Dump via an Explicit Action by the Customer

A customer can explicitly cause a thread dump of the JVM process through an operating system command or the Task Manager in Windows.

- [Triggering a javacore using an operating system command](#)  
Issuing a kill statement against the JVM process on a Unix system typically causes a thread dump; however, that depends on the signal that is specified on the command, and if there is a signal handler that intercepts that signal before the JVM gets it. The JVM process is not always ended by an operating system signal; that depends on the signal that is sent to the JVM process. A kill -3 (SIGQUIT) is not a terminating signal. A kill -11 (SIGSEGV) is a terminating signal. The kill command on AIX, by default, sends a SIGTERM signal.
- [Triggering a javacore using a WebSphere Application Server script or bat file](#)  
For V3.5 and V4.0 releases on Windows, the DrAdmin.bat utility also causes a thread dump. See [WebSphere V3.5.x + V4.0.x: DrAdmin](#) for more information.

For V5.0 releases, wsadmin can be used to create a Java thread dump. For additional information, see technote [MustGather: No response \(hang\) or performance degradation on Windows for v5.0 releases](#).

**Note:** Using these utilities to dump threads does not end the JVM process.

[back to top](#) ↑

## 3. Reading javacore files

Javacore files are text files, so you can read them with a text editor. There are two different formats of javacore files. The older format is simpler. This document provides examples and discusses both formats. The examples refer to javacores on AIX systems; however, the principles are applicable to Linux and Windows IBM Java SDKs.

[back to top](#) ↑

### 3.1 Signal information

This section tells you details about the operating system signal that produced the javacore. It identifies when the signal was issued and where the signal was issued.


#### 3.1.1 Time and Date of the javacore

Every javacore file has a line that indicates the time and date of when the javacore file was produced. In the older javacore format, the first line of a javacore file identifies when the javacore was taken. In the newer javacore format, there is a line that has the tag, 1TIDATETIME and the tag Date: that

indicates the date and time when the javacore was written.

#### ***Diagnostic Step***

Verify when the javacore file was written to determine if it is relevant to problem being analyzed.

[back to top](#) 

#### **3.1.2 Type of Signal and Location of the JVM process at the time of the javacore**

Every javacore has a line that indicates the signal that caused the javacore to be produced. In the older javacore format, this is the next line after the date in the javacore file. In the older javacore file format, the line that identifies the signal often identifies the library in which the signal was generated (if it is possible for the javacore processing to identify it). In the sample javacore output shown, the signal that generated the thread dump is a SIGSEGV, which means that the JVM process ended abnormally. The library in which the crash occurred is libdb2jdbc.so.

#### **Sample header from the old javacore format**

```
Tue Mar 9 18:18:31 2004
SIGSEGV received at 0xd44350a8
in /usr/lpp/db2_07/java12/libdb2jdbc.so.
Processing terminated.
J2RE 1.3.1 IBM AIX build ca131-20020722
```

In the newer javacore file format, the line with the tag 1TISIGINFO contains the signal that caused the javacore. It is important to understand what signal caused the javacore. This determines how to interpret the remaining information in the javacore file.

For the newer javacore format, the library that issued the operating system signal is identified in the line with the tag 1XHSIGRECV.

#### **New format javacore**

```
NULL -----
0SECTION      TITLE subcomponent dump routine
NULL          =====
1TISIGINFO     signal 11 received
1TIDATETIME    Date:                      2004/04/06 at 16:04:16
1TIFILENAME     Javacore filename:
                /export/home/was4/wasapps/appvmc01/vmclaimt1/workdir/javacore4
NULL          -----
0SECTION      XHPI subcomponent dump routine
NULL          =====
1XHTIME        Tue Apr 6 16:04:16 2004
1XHSIGRECV     SIGSEGV received at 0xd2782a88
in /opt/WebSphere4/AppServer/java/jre/bin/libjipc.a. Processing
1XHFULLVERSION J2RE 1.3.1 IBM AIX build ca131-20030630a
```

#### ***Diagnostic Step: Examining the Signal type and location***

First, note the signal type. Does the signal indicate a JVM crash? Does the signal information state it was due to an operator action? If the javacore is written to diagnose a hung JVM process, the steps you take are very different than if the javacore is due to a JVM crash.

Next, if it is a signal that indicates a JVM crash, examine this line to view the signal that caused the javacore to be written and to see if it identifies the library that caused the JVM crash. Contact the company that developed the library to pursue a fix. If the library is a JVM native library, a Java SDK upgrade might resolve the problem. If the signal information does not identify the library that caused the crash, you must examine the Current Thread Details to see if you can use determine the library that caused the crash.

For example, the following message indicates a crash in a JVM library and should be pursued with WebSphere Application Server Support.


#### **Crash in the libjipc.a library**

```
1XHSIGRECV     SIGSEGV received at 0xd2782a88
in /opt/WebSphere4/AppServer/java/jre/bin/libjipc.a.
Processing terminated.
```

This crash shows a JVM crash in a DB2® library and should be pursued with DB2 support.

#### Crash in a DB2 runtime library

```
SIGSEGV received at 0xd44350a8
in /usr/lpp/db2_07_01/java12/libdb2jdbc.so.
Processing terminated.
```

[back to top](#) 

#### 3.1.3 Java Fullversion

The javacore identifies the Java fullversion of the JVM that produced the javacore. In the older javacore format, this line is the one after the one with the operating system signal. The line in the example above that reads J2RE 1.3.1 IBM AIX build ca131-20020722. In the new javacore format, the Java fullversion is in the line with the tag 1XHFULLVERSION.

[back to top](#)

#### 3.1.4 Current Thread Details

The current thread is the thread that is running when the signal is raised that causes the javacore to be written. The **Current Thread Details** shows the Java and native stacks of the current thread. Since these are stacks, the most recent calls are at the top of the stack.

- The Java stack shows the Java method calls that are made by the current thread.
- If there is a native stack, this contains the most recent events that the thread executed. As the name implies, these are events in native code (libraries) called by Java.

#### Diagnostic Step

If the library is not identified by the signal information, you must examine the current thread details to see if the native stack indicates in which library the current thread was processing at the time of the JVM crash. If the signal is one that is used for an abnormal process termination, that is, SIGSEGV or SIGILL, the current thread details show the sequence of calls that caused the signal to be generated. For a JVM crash, if the line that identifies the signal is not able to identify the failing library, the native stack trace in the current thread details might be useful to identify the library.

In the two examples here, the location of the crash was already determined by the javacore processing, but this is not always the case.

#### Example 1

This is the current thread for the SIGSEGV in libdb2jdbc.so shown above. The current thread shows none of the Java stack; it might be that the javacore processing could not retrieve the information for the Java stack. The native stack shows that a jni\_NewString call was made that caused the SIGSEGV to occur; the call to jni\_NewString was made from Java\_COM\_ibm\_db2\_jdbc\_app\_SQLExceptionGenerator\_SQLStatementError, which is a DB2 Java method.

#### Current Thread Details

```
-----
"Servlet.Engine.Transports:253" sys_thread_t:0x7D4C36C8
----- Native Stack -----
at 0xD3FF7A34 in jni_NewString
at 0xD44350A8 in
Java_COM_ibm_db2_jdbc_app_SQLExceptionGenerator_SQLStatementErr
```

#### Example 2

In this example, the current thread is in the libjitc.a library, which is generating native code for a Java method. This is not obvious from the first three lines of the native stack, but the fourth call down shows that it is the JIT compiler is involved.

```
1XHCURRENTTHD Current Thread Details
NULL -----
2XHCURRSYSTHD "EntigoAppsStarter"
sys_thread_t:0x59AF8650
3XHNATIVESTACK Native Stack
```

```

NULL -----
3XHSTACKLINE   at 0xD2782A88 in
dataflow_arraycheck
3XHSTACKLINE   at 0xD27226A0 in
bytecode_optimization_driver
3XHSTACKLINE   at 0xD27251CC in
bytecode_optimization
3XHSTACKLINE   at 0xD2685140 in
JITGenNativeCode
3XHSTACKLINE   at 0xD26AB774 in
jit_compile_a_method_locked
3XHSTACKLINE   at 0xD26ACD24 in
jit_compiler_entry
3XHSTACKLINE   at 0xD26AD284 in
_jit_fast_compile

```

In this case, a workaround is to skip JIT compiling of this method. A likely resolution is to upgrade the Java SDK to upgrade the `libjitr.a` library.

**Note:** For a JVM crash, a formatted core file or user.dmp file is also very helpful for diagnosing the cause of the problem.

[back to top](#)

### 3.2 Operating Environment

This section provides details about the operating system and processor (system) that the JVM is running on. This section might also contain information about paging, user limits (rlimits), and environment variables that are set.

[back to top](#)

#### 3.2.1 Signal Handlers

This section shows you the Signal Handlers and the Environment Variables that are in use by the JVM process. A signal handler is a library that is defined to handle a signal if and when it occurs. These can be programmatically defined; a library can register itself to handle a signal, intercept it and process it, and prevent it from being handled by the JVM process.

#### Signal Handler Info in new javacore format

```

1XHSIGHANDLERS Signal Handlers
NULL -----
2XHSIGHANDLER SIGHUP   : ignored
2XHSIGHANDLER SIGINT   : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGQUIT  : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGILL   : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGTRAP  :
JITSigTrapHandler (libjitr.a)
2XHSIGHANDLER SIGABRT  : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGEMT   : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGFPE   : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGBUS   : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGSEGV  : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGSYS   : intrDispatchMD
(libhpi.a)
2XHSIGHANDLER SIGPIPE  : ignored
2XHSIGHANDLER SIGUSR2  :
JITSigUSR2Handler (libjitr.a)

```

```
2XHSIGHANDLER SIGTERM : intrDispatchMD
(libhpi.a)
```

**Diagnostic Step**

You can use the signal handler information to determine if a signal handler is intercepting a signal and preventing a javacore from being produced.

**Note:** libhpi.a is the normal JVM library for handling signals on AIX and Linux; it is the library that produces the javacore.

If a signal handler is preventing the javacore from being produced, then the wscp or wsadmin utility might be used to dump the threads and get a javacore. Furthermore, signal handlers can be disabled for the sake of diagnosing a problem. If you know that a product uses a signal handler, as IBM MQ® does, you can find out from product documentation how to disable the signal handler.

[back to top](#)

**3.2.2 Loaded Libraries**

This section of the javacore shows the native libraries that are loaded by the JVM.

**Loaded library output for the libjtc.a library**

```
2XHLIBNAME /opt/WebSphere4/AppServer/java/jre/bin/libjtc.a
3XHLIBSIZE      filesize      : 2731317
3XHLIBSTART     text start   : 0xD264A000
3XHLIBLDSE      text size    : 0x233D6C
3XHLIBLDORG     data start   : 0x524AE498
3XHLIBLDATASZ   data size    : 0xF71C
```

**Diagnostic Step**

This can be useful in determining which version of a library is loaded in the event that there is more than one copy of the library installed on a system.

It can also be used to determine the location of a crash if the library name is not identified. This requires looking at the **Text start** and **text size** lines. In the new format javacore attached to this document, the offset of the crash is 0xd2782a88. The library whose text range contains the address offset is the library in which the signal occurred that produced the javacore. For the libjtc.a, its text range is 0xD264A000 to 0xD287DD6C.

[back to top](#)

**3.2.3 Java Command Line**

The javacore contains the command line that started Java. In the older javacore format, this is in a section entitled, System Properties. In the newer javacore format, the command line that started Java is identified with the tag 1CICMDLINE. This is in the CI subcomponent dump routine.

Use the Java command line to determine if the javacore is for a WebSphere Application Server Java process, and if so, which WebSphere Java process it is for: an Application Server process, an Administrative Server process, a jmsserver, a node agent, or some other WebSphere Application Server process.

[back to top](#)

**3.2.3.1 Determining which WebSphere Application Server process a javacore is for**

One way you can determine which Java process the javacore is for is to use the process ID of the javacore and match that to the process ID in the WebSphere Application Server logs. For example, the startServer.log or the SystemOut.log shows the process ID that the WebSphere Application Server Java process is using.

You can also determine which WebSphere Java process the javacore is for by finding the class that was run when the JVM process was started. This must be found in the actual command line. The class name appears in the Java command line by itself, without any -, -D, or -X preceding it. Use the following tables to assist you when associating a WebSphere Application Server Java process with a javacore.

**For V4.0 releases:**

WebSphere Component	Component Class Name
AppServer	com.ibm.ejs.sm.server.ManagedServer
AdminServer	com.ibm.ejs.sm.AdminServer
Nanny process	com.ibm.ejs.sm.Nanny

wscp	com.ibm.ws.bootstrap.WSLauncher <b>com.ibm.ejs.sm.ejscp.WscpShell</b>
------	--

For V5.0 and V5.1 releases, it always shows com.ibm.ws.bootstrap.WSLauncher as the class that is started by Java; however, it contains the class listed under Component Class Name in the Java command line for the component listed.

WebSphere Component	Component Class Name
AppServer	com.ibm.ws.management.tools.WsServerLauncher with com.ibm.ws.runtime.WsServer in the command line
nodeagent	com.ibm.ws.management.tools.WsServerLauncher with nodeagent in the command line
jmsserver	com.ibm.ws.management.tools.WsServerLauncher with jmsserver in the command line
wsadmin	com.ibm.ws.scripting.WasxShell

[back to top](#)

### 3.3 JVM monitor information

The monitor information shows what synchronization locks are held by which threads. It also shows which threads are blocked by monitors. This information is useful for determining the cause of a deadlocked or hung JVM. In the old javacore format, the monitor information is in a section entitled, LK component Dump Routine. This is written after the thread dump of each thread in the JVM. In the new javacore format, the monitor information is in a section entitled LK subcomponent dump routine. It is before the thread dump of all the threads of the JVM.

For analyzing a potential deadlock, the [thread analyzer](#) tool can be very helpful. Using the javacore file to determine a potential deadlock, the JVM code that produces the javacore does examine the monitor information for deadlocks and does try to identify potential deadlocks. It can also be useful to check monitors to see if there are some on which many threads are blocked. You still have to confirm that there is an actual deadlock. A large number of threads blocked on a monitor does not mean a deadlock has occurred. It might mean that there is a monitor (synchronization lock) that is causing a backlog of work to be completed.

[back to top](#)

### 3.4 The JVM thread dump

The javacore processing dumps the current stack for every thread in the JVM. It shows the current state of the thread, that is whether it is *Runnable* or not. It also shows the thread name, which is useful in determining how that thread is used. It also shows the Java stack and native stack for each thread.

Here is a sample thread dump for a Servlet.Engine.Transports thread. This shows that this thread is in Runnable state (state:R), and it shows the thread ID. This information can be used to correlate this thread information with other views of threads in the Java process.

#### Thread dump of a Servlet.Engine.Transports thread

```
"Servlet.Engine.Transports:239" (TID:0x34B94018,
sys_thread_t:0x7CD4E008, state:R, native
ID:0x10506) prio=5
at
java.net.SocketInputStream.socketRead(Native
Method)
at
java.net.SocketInputStream.read
(SocketInputStream.java(Compiled Code))
at
com.ibm.ws.io.Stream.read(Stream.java(Compiled
Code))
at
com.ibm.ws.io.ReadStream.readBuffer
(ReadStream.java(Compiled Code))
at
com.ibm.ws.io.ReadStream.read(ReadStream.java
```



```

(Compiled Code))
at
com.ibm.ws.http.HttpConnection.run
(HttpConnection.java(Compiled Code))
at
com.ibm.ws.util.CachedThread.run
(ThreadPool.java:137)
----- Native Stack -----
at 0xD41D0E58 in sysTimeout
at 0xD4012A30 in JVM_Timeout
at 0xD4424C54 in
Java_java_net_SocketInputStream_socketRead

```

**Thread names**

The thread name is useful in determining what process owns the thread and how that thread is used. The following table shows some common thread names, what product uses this thread, and what the thread is used for.

Thread name	JVM Process	Purpose
Alarm Thread #	Releases of v4.0 and v5 Application Servers	handles timer processing
Session.Transports.Threads:###	Releases of v4.0 and v5 Application Servers	servlet threads for processing HTTP requests
ORB.thread.pool:###	Releases of v4.0 and v5 Application Servers	an ORB thread used for sending ORB data
P=437206:O=0: StandardRT=19027: LocalPort=9001: RemoteHost=hostname.ibm.com:RemoteP	Releases of v4.0 Administrative Server and releases of v4.0 and v5 Application Servers	an ORB thread for receiving an EJB request or other ORB request
Thread-##	JVM thread	thread created by the JVM; this is the default thread name.
Finalizer	JVM thread	used to run finalize methods in Java objects.
PingThread	Releases of v4.0 Application Servers	thread used for ping processing with Administrative Server
Alarm Manager	Releases of v4.0 and v5 Application Servers	Manages alarm threads
SoapConnectorThreadPool : #	Releases of v5 Application Servers and JMS server	Soap thread pool thread
BrokerDFEThread	Releases of v5 Application Servers	MQ Broker thread
GC Daemon	Any JVM	Java GC daemon thread
GCHelper#	Any JVM	Java GC helper thread
LT=0:P=70863:O=0:port=62111	Releases of v4.0 Administrative Server and	a local Orb connection receiver thread

	releases of v4.0 and v5 Application Servers	
java.net.MulticastSocket@11b79f6	Releases of v5 Application Servers	Discovery management thread
SSConnection	Releases of v5 Application Servers	Server socket connection for JMS server
DynaTopology	Releases of v5 Application Servers	Dynamic topology thread
MQQueueAgent	Releases of v5 Application Servers	MQ Queue Agent thread

**Thread states**

The thread state indicates if the thread is currently runnable or not. If the thread state is state:R, the thread is runnable. The following thread states indicate a thread that is in a wait state: state:CW (Conditioned Wait) and state:MW (MuxSemWait).

**Java stack**

The call stack under the thread header is the Java stack. This shows the Java calls that have been made to get the thread to its current state. The first line in the Java stack is the last Java method call that was made. It was from that location that a call into a native method might have been made. That is typically identified with the phrase **Native Method** showing the location in the Java program that was called.

**Native stack**

The native stack shows what native methods (procedures) were called after the thread entered the native code. The first line in the native stack shows what the thread was doing in native code when the javacore was taken.

**CL subcomponent dump routine section**

The new javacore format also contains a section that dumps the classloader information for the Java system classloaders.

[back to top](#)

**4. How to proceed if the library that caused the crash is not identified**

Frequently, the javacore does not clearly identify the cause of the signal. And often the Native Stack is not listed plainly as it is above. Often the Native Stack will show the following:

```
----- Native Stack -----
unable to backtrace through native code - iar
0x3062e73c not in text area (sp is 0x2ff21748)
```

At this point there are a few steps you can take.

1. Disable JIT compilation. Frequently when the crash is not identified in the javacore, it is due to a problem with the JIT compiled code. Disabling JIT compilation can be used to confirm if this is the case. The JIT compiler is a JVM optimization. There can be significant performance degradation when the JIT is disabled, as much as 20% or more. To disable the JIT compiler, refer to [Disabling the Just-In-Time \(JIT\) compiler in WebSphere Application Server V5.0 and V5.1 releases](#) or [Disabling the Just-In-Time \(JIT\) compiler in WebSphere Application Server V4.0](#).

If the problem is in the JIT compiler, you might be able to skip JIT compiling for a method if the problem is in the code generated for a method or in the code generation itself.

Refer to the technote [Using the JTC\\_COMPILEOPT=COMPILING variable to skip a failing method](#) to skip JIT compiling for a specific method.

You can also use technote [Selectively disabling JIT options for WebSphere Application Server V3.5, V4.0, V5.0 and V5.1](#) to disable certain parts of the JIT compiler to prevent the problem.

2. Upgrading to a more recent JDK can sometimes resolve a problem. To download the latest JDK packaged for WebSphere Application Server, refer to the [Recommended Updates](#) page for WebSphere Application Server.
3. Use the core file (on UNIX) or `user.dmp` file (on Windows) to see if this provides more information. These files are binary and must be formatted. For tools to format these files, refer to these technotes:
  - [MustGather: Crash on Microsoft Windows](#)
  - [MustGather: Crash on AIX](#)

- [MustGather: Crash on Linux](#)

4. Sometimes a bad Java SDK installation can cause problems. The fullversion for the javacore file comes from the libjvm.a. The fullversion for the java -fullversion command is from the Java executable. If there is a discrepancy between these two dates, either the wrong libjvm is picked up due to an error in the library path statement, or there is a problem in the JVM installation.

[back to top](#)

#### 5. What can prevent a javacore file from being produced?

A signal handler from another product can prevent a javacore from being produced. In this case, the defined signal handler intercepts the signal before it gets to the Java library that handles the signals and produces the javacore, and that signal handler does not forward the signal to the JVM for processing.

Setting the `DISABLE_JAVADUMP=true` can prevent a javacore from being produced.

Setting `IBM_NOSIGHANDLER=true` can prevent a javacore from being produced.

In some cases a hung process cannot produce a javacore file. On rare occasions, the javacore processing is the cause of this. In these situations, it is necessary to disable the javacore processing and use a process dump (core file on Unix and user.dmp file on Windows) to diagnose the problem. For more details, refer to the following technotes:

- [MustGather: Getting user.dmp when Hangs/Performance Degradation prevents generating a javacore](#)
- [MustGather: Crash on AIX produces no core or a truncated core](#)

For more information on the IBM Java developer kits, refer to the [IBM developer kits - diagnostic guides](#).

[back to top](#)