POONAM BAJAJ

BIO

HotSpot's Hidden Treasure

The HotSpot Serviceability Agent's powerful tools can debug live Java processes and core files.

The HotSpot Serviceability Agent is a hidden treasure present in the JDK that very few people know about. The Serviceability Agent (SA) is a set of Java APIs and tools that can be used to debug live Java processes and core files (also called *crash dumps* on Microsoft Windows).

SA can examine Java processes or core files, which makes it suitable for debugging Java programs as well as the Java HotSpot VM. It is a snapshot debugger and lets us look at the state of a frozen Java process or a core file. When SA is attached to a Java process, it stops the process at that point and we can explore the Java heap; look at the threads that were running in the process at that point; examine internal data structures of the Java HotSpot VM; and look at the loaded classes, compiled code of methods, and so on. The process resumes after SA is detached from it.

SA Binaries

Before we go into the details about the features and utilities that SA offers, I would like to mention SA binaries that are present in the JDK. There are two SA binaries that are shipped with the JDK:

- For Microsoft Windows: sa-jdi.jar and jvm.dll
- For Oracle Solaris and Linux: sa-jdi.jar and libsaproc.so

These binaries provide the SA Java APIs and also include useful debugging tools implemented using these APIs.

JDK Versions with Complete SA Binaries

The following JDK versions have complete SA binaries:

- JDK 7 on all platforms
- 6u17+ on Oracle Solaris and Linux
- 6u31+ on Microsoft Windows

Prior to these versions, SA was not shipped with the JDK on Microsoft Windows, and only a subset of SA classes was shipped with JDKs on Oracle Solaris and Linux. The JDK versions above make the complete set of SA classes available on all of these platforms.

DETAILS, DETAILS

SA makes it very easy to **examine the Java-level details and JVM-level details** of a Java process or core file.

Why Use SA?

Why use SA when we have native debugging tools such as dbx, GDB, WinDbg, and many others?

First, SA is a Java-based, platform-independent tool, so it can be used to debug Java processes and cores on all the platforms where Java is supported. Additionally, debugging a Java process or the Java HotSpot VM with native debuggers is very limiting, because although native debuggers can help us examine the native OS process state, they cannot help us examine the Java or the Java Virtual Machine (JVM) state of the process.

For example, if I need to view the objects in the Java heap, native debuggers would show me the raw hex numbers, whereas SA has the ability to interpret those hex numbers and present the object view instead. SA has knowledge about the Java heap, such as its boundaries, objects in the Java heap, loaded classes, thread objects, and internal representations of the Java HotSpot VM. SA makes it very easy for us to examine the Java-level details and JVM-level details of the Java process or core file.

SA Debugging Tools

There are two main SA debugging tools implemented using SA APIs:

- HSDB, which is a GUI tool and the main debugger
- CLHSDB, which is a command-line variant of HSDB

HSDB: The GUI debugger. HSDB facilitates examining Java processes, core files, and also remote Java processes. Let's see how we can launch and use it on a Microsoft Windows machine.

First, we need to set some environment variables. Set SA_JAVA to the location of the Java executable in the JDK/bin folder, for example:

```
set SA_JAVA=
d:\java\jdk1.7.0_03\bin\java
```

On Microsoft Windows, the PATH environment variable should contain the location of the JVM binary used by the target pro-

cess or core and also the folder where the Debugging Tools for Windows are installed on the machine, for example:

```
set PATH= d:\java\jdk1.7.0_03\bin\
server;d:\windbg;%PATH%
```

Set the PATH environment variable and then launch HSDB as follows:

```
java -Dsun.jvm.hotspot.debugger.
useWindbgDebugger=true -classpath
d:\java\jdk1.7.0_03\lib\sa-jdi.jar
sun.jvm.hotspot.HSDB
```

On an Oracle Solaris or Linux machine, we just need to set SA_JAVA to the Java executable and then we can launch HSDB as follows:

```
java -Dsun.jvm.hotspot.debugger.
useProcDebugger=true -classpath
/java/jdk1.7.0/lib/sa-jdi.jar
sun.jvm.hotspot.HSDB
```

These launch commands bring up the HSDB GUI tool, as shown in **Figure 1**.

Let's take a quick look at some of the very useful utilities available in this tool, which are shown in **Figure 2**.

Figure 3 shows the Object Inspector, which you can use to inspect Java objects.

Figure 4 shows how you can find where a particular address lies in the Java process.

Figure 5 shows the Object Histogram.

You can find the heap boundaries, as shown in **Figure 6**.

CLHSDB: The command-line debugger. CLHSDB is the command-line variant of HSDB.

We need to set the same environment variables for CLHSDB as we did for HSDB. Use the following command to launch this tool on Microsoft Windows:

```
java -Dsun.jvm.hotspot.debugger.
useWindbgDebugger=true -classpath
d:\java\jdk1.7.0_03\lib\sa-jdi.jar
sun.jvm.hotspot.CLHSDB
```

CLHSDB offers almost all the features that the GUI version of the tool offers.

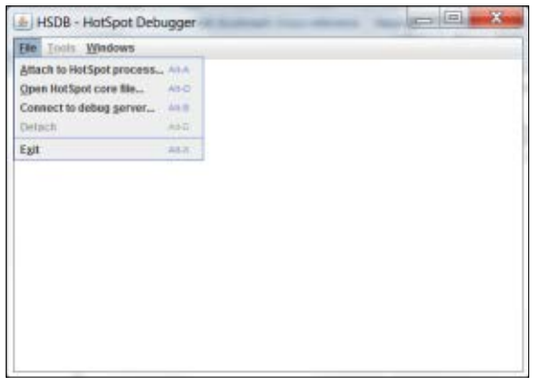

Figure 1

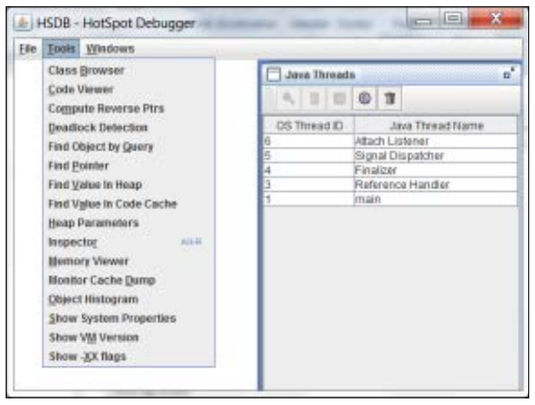

Figure 2

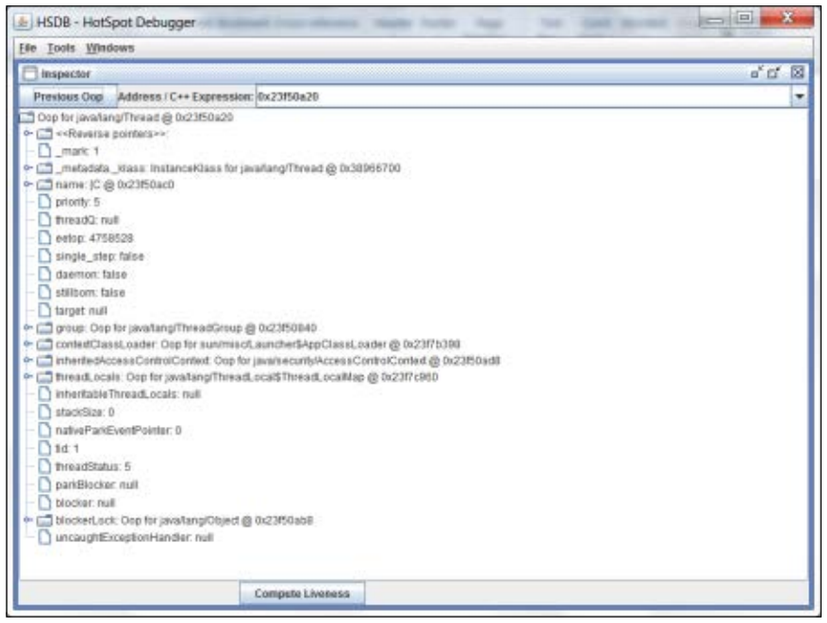

Figure 3

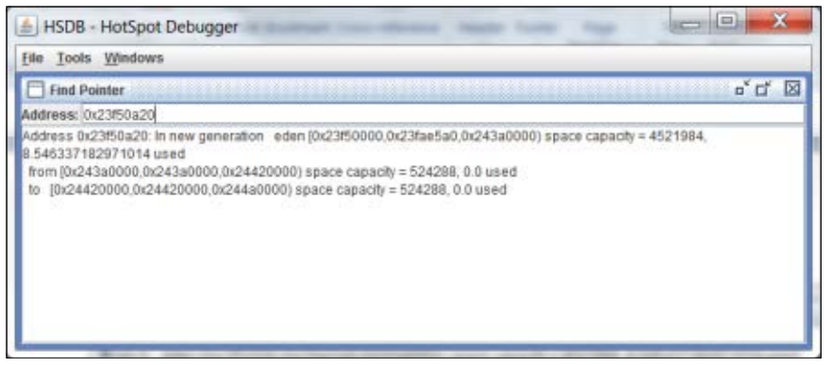

Figure 4

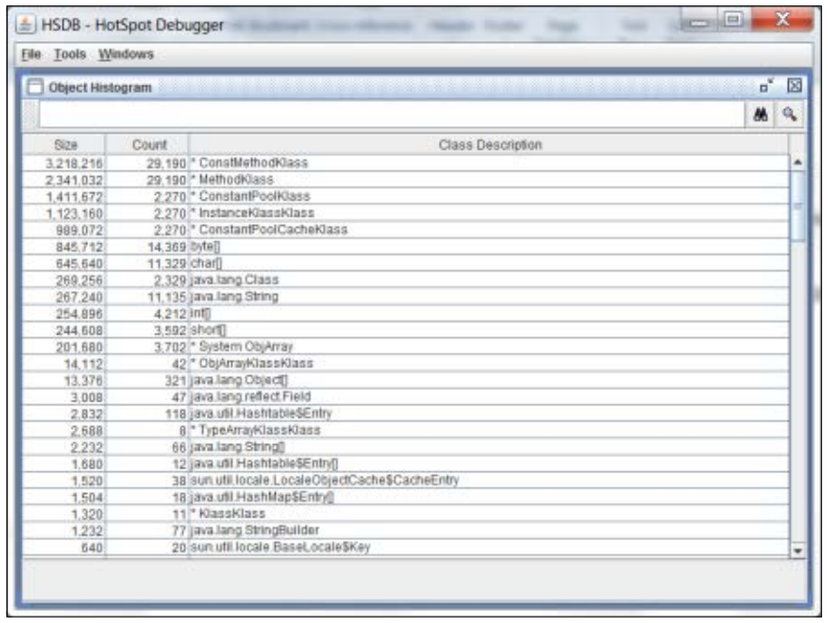

Figure 5

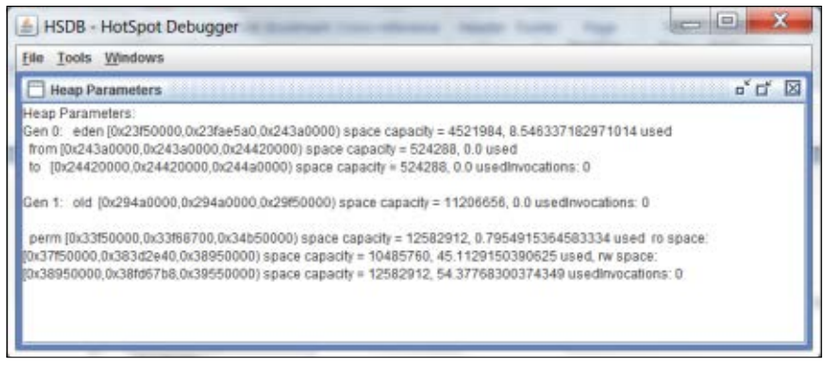

Figure 6

For example, to examine any Java object, use the inspect command, as shown in **Listing 1**.

To look at heap boundaries, we can use the universe command, as shown in **Listing 2**.

Listing 3 and **Listing 4** show the complete list of commands available with this tool.

Other Tools

There are some other very handy small utilities bundled with SA. Let's see how to use them and how their output looks:

- FinalizerInfo prints details on the finalizable objects, as shown in **Listing 5**.
- HeapDumper dumps the heap in HPROF format, as shown in **Listing 6**.
- PermStat prints the permanent generation statistics, as shown in **Listing 7**.
- PMap prints the process map of the process (see **Listing 8**), much like the Oracle Solaris pmap tool does.
- SOQL, the Structured Object Query Language tool, is an SQL-like language that we can use to query the Java heap, as shown in **Listing 9**. JHat also provides an interface for using this language, and pretty good documentation on this language is also available in JHat.
- JSDB, the JavaScript Debugger, provides a JavaScript interface to SA (see **Listing 10**). It is a command-line JavaScript shell based on Mozilla's Rhino JavaScript engine. More details on this utility can be found in the open source Java HotSpot VM repository in the file hotspot/agent/doc/jsdb.html.

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
hsdb> inspect 0x23f50a20
instance of Oop for java/lang/Thread @ 0x23f50a20 @ 0x23f50a20 (size = 104)
_mark: 1
_metadata._klass: InstanceKlass for java/lang/Thread @ 0x38966700 Oop @
0x38966700
name: [C @ 0x23f50ac0 Oop for [C @ 0x23f50ac0
priority: 5
threadQ: null null
eetop: 4758528
single_step: false
daemon: false
stillborn: false
target: null null
group: Oop for java/lang/ThreadGroup @ 0x23f50840 Oop for java/lang/Thread-
Group @ 0x23f50840
contextClassLoader: Oop for sun/misc/Launcher$AppClassLoader @ 0x23f7b398 Oop
for sun/misc/Launcher$AppClassLoader @ 0x23f7b398
inheritedAccessControlContext: Oop for java/security/AccessControlContext @
0x23f50ad8 Oop for java/security/AccessControlContext @ 0x23f50ad8
threadLocals: Oop for java/lang/ThreadLocal$ThreadLocalMap @ 0x23f7c960 Oop
for java/lang/ThreadLocal$ThreadLocalMap @ 0x23f7c960
inheritableThreadLocals: null null
stackSize: 0
nativeParkEventPointer: 0
tid: 1
threadStatus: 5
parkBlocker: null null
blocker: null null
blockerLock: Oop for java/lang/Object @ 0x23f50ab8 Oop for java/lang/Object @
0x23f50ab8
uncaughtExceptionHandler: null nullCheck heap boundaries
```

Download all listings in this issue as text

Let's Get Our Hands Dirty

Let's get a real feel for the SA tools and debug a Java program crash using them. I have a simple program of Java Native Interface (JNI) code that writes to a byte array beyond its size limit, which results in overwriting and corrupting the object that follows it in the Java heap. This causes the program to crash when the garbage collector tries to scan the heap. See **Listing 11**.

The crash happened in objArrayKlass::oop_follow_contents(oopDesc*) at program counter

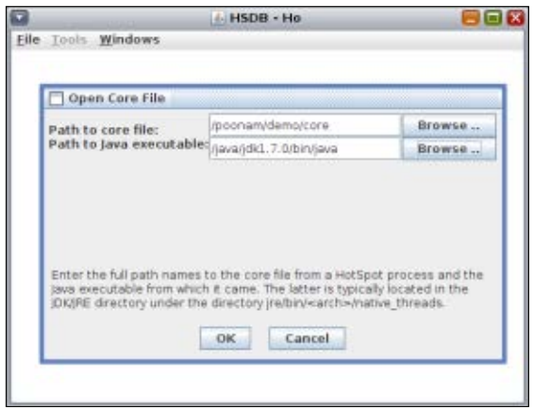

Figure 7

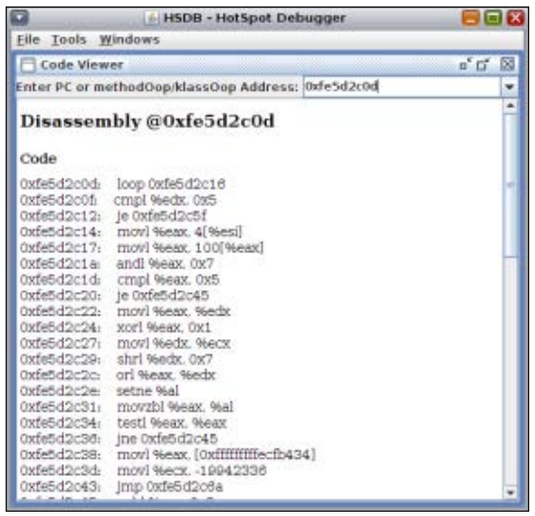

Figure 8

(PC) 0xfe5d2c17. See **Listing 12**, which shows the stack trace of the crash from the hs_err file.

With the crash, a core file got generated. Let's open this core with HSDB (see **Figure 7**), dig out some information from it, and try to find the cause of this crash.

Figure 8 shows the disassembly of the code that was being executed around PC 0xfe5d2c17 when the crash happened.

The instructions shown in **Figure 8** indicate that the process crashed when trying to access the value at address eax+100. From the hs_err file, we can

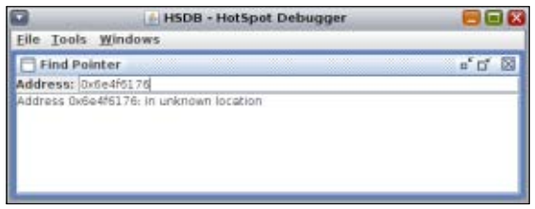

Figure 9

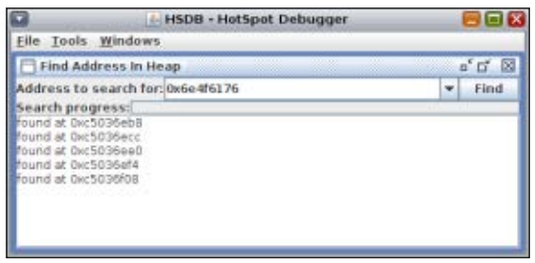

Figure 10

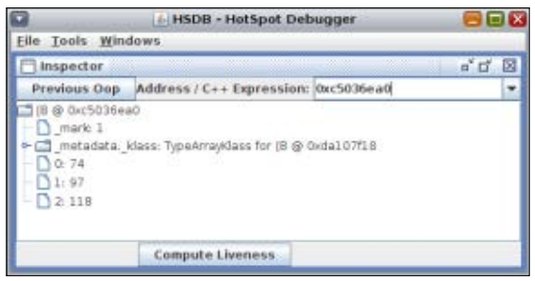

Figure 11

see the contents of the registers and what the value of the EAX register was:

> **EAX=0x6e4f6176,** EBX=0xc50a083c, ECX=0x614a2e2e, EDX=0x00000006 ESP=0xfbc7e360, EBP=0xfbc7e398, ESI=0xc5036ef0, EDI=0x00000000 EIP=0xfe5d2c17, EFLAGS=0x00010202

What was at 0x6e4f6176, and why did the crash happen while reading the value at this address? HSDB helps us see that, as shown in **Figure 9**.

The address does not lie in the Java heap. Using the **Find Address in Heap** option, we can find the locations in the Java heap from which this particular address is referenced (see **Figure 10**).

Examine these found locations in the Object Inspector to see if these are part of any object, as shown in **Figure 11**.

All the found addresses bring up the byte array object at 0xc5036ea0 in the Object Inspector, which means the object at 0xc5036ea0 is the closest valid

LISTING 7 LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11 / LISTING 12

```
java -Dsun.jvm.hotspot.debugger.useWindbgDebugger=true -classpath d:\java\
jdk1.7.0_03\lib\sa-jdi.jar sun.jvm.hotspot.tools.PermStat 5684
Attaching to process ID 5684, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 22.1-b02
10713 intern Strings occupying 802608 bytes.
finding class loader instances ..
done.
computing per loader stat ..done.
please wait.. computing liveness.....done.
class_loader   classes bytes  parent_loader  alive?  type

<bootstrap>   342   1539808 null      live   <internal>
0x23f7b398   3    28016  0x23f762e0    live   sun/misc/Launcher$AppClassLoader
@0x38a0e9c0
0x23f762e0   0    0      null      live   sun/misc/Launcher$ExtClassLoader@0x389
eb420

total = 3    345   1567824   N/A      alive=3, dead=0   N/A
```

Download all listings in this issue as text

LISTING 13 LISTING 14

```
(dbx) x 0xc5036ea0/100c
0xc5036ea0:   '\001' '\0' '\0' '\0' '\030' '\0177' '\020' '□' '\003' '\0' '\0' '\0'
'H' 'e' 'l' 'l'
0xc5036eb0:   'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a' 'v'
0xc5036ec0:   'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e' 'l'
0xc5036ed0:   '\003' '\0' '\0' '\0' 'a' 'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a'
0xc5036ee0:   'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e'
0xc5036ef0:   'l' 'l' 'o' ' ' 'J' 'a' 'v' 'a' '.' 'H' 'e' 'l' 'l' 'o' ' ' 'J'
0xc5036f00:   'a' 'v' 'a' '.'
```

Download all listings in this issue as text

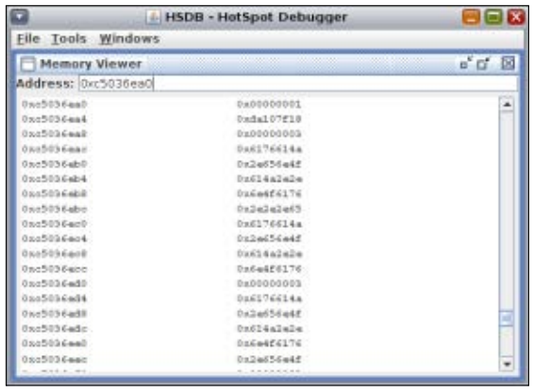

Figure 12

object just before these locations. If we look carefully, these locations actually go beyond the limits of the byte array object, which should end at 0xc5036eb0, and from address 0xc5036eb0, the next object should have started. See the raw contents at memory location 0xc5036ea0 in **Figure 12**.

We can look at the raw contents as characters in the dbx debugger. See **Listing 13**, which clearly shows that the object at 0xc5036ea0 has a byte stream that goes beyond its size limit of three elements and overwrites the object starting at 0xc5036eb0.

This gives us a big clue. Now, we can easily search in the code where the bytes "Hello Java.Hello Java. . ." are being written, and find the buggy part of the code that overflows a byte array. **Listing 14** shows the faulty lines that I had in my JNI code. Wow! This was so easy.

Summary

As in the example above, we in the JVM Sustaining Engineering Group at Oracle use the Serviceability Agent on a daily basis to debug crashes, hangs, and other kinds of problems that occur with the Java HotSpot VM. SA is a pretty useful and powerful debugging tool that can also help you learn the internals of the Java HotSpot VM. I hope this article provided good insight into this tool. Enjoy debugging with SA! `</article>`

LEARN MORE

- SA-Plugin for VisualVM