WebSphere® software

Redbooks Paper

Rica Weller
Dave Clarke
Dave Griffiths

WebSphere for z/OS V5 JVM dump and heap analysis tools

Introduction

This IBM Redpaper describes the following problem analysis tools for the IBM WebSphere® for z/OS® production environment:

- Svcdump.jar
- HeapRoots
- Dumpviewer GUI and jformat

To analyze heap-related issues such as OutOfMemoryError and other crashes, as well as hangs or loops within WebSphere address spaces with a level of degree similar to that of older deployment environments such as CICS® and IMS™, use the relatively low impact tool of unformatted dumps and Java™ virtual machine (JVM) internal information.

This Redpaper goes into great detail about the switched virtual circuit (SVC) Analyzer, explaining what svcdump.jar is, where to get it and when and how to use its utilities. Finally, we list several samples for different parameter settings.

The HeapRoots utility and the DumpViewer GUI are mentioned for further information.

Why use these tools?

The Java virtual machine (JVM) sits at the center of the execution environment for WebSphere. The many benefits of using Java as a development language and a JVM as a targeted runtime come with the knowledge that diagnosis of problems in a production environment can seem both more difficult than and very remote from the original development environments, where real-time, GUI-based debugging is the norm.

From a technical perspective, this real-time approach can transfer to IBM @server zSeries® and production servers. It is possible to load a server with a JVM in debug mode and attach a remote debugger such as WebSphere Studio Application Developer, Integration Edition. In addition, tools mentioned in this paper make use of the various JVM interfaces: JVM Profiling Interface (JVMPI), JVM Monitoring Interface (JVMMI), and so on, to provide real-time monitoring and profiling of the JVM attached in WebSphere servant processes.

These techniques work well when:

- Transaction throughput is low
- Targeting behavior on a server used for z/OS development
- Testing before deployment to a production server

They are usually unsuitable for production for a variety of reasons:

- They require you to load a debug version of the JVM. The debug version is built with many asserts for invalid conditions enabled. This means the performance of the JVM is not adequate for use in production.
- The quantity of data generated by profiling tools from a server with a high transaction throughput is generally much larger than can be accommodated by the system, or meaningfully post-processed from the profiling tool.
- Profiling tools are typically fairly independent of platforms and, as a result, not sensitive enough to platform peculiarities or problems that need to be solved across middleware or OS components.

These issues mean that we need to be able to acquire knowledge of the JVM internals with less invasive diagnostic approaches, such as SDUMP, than are typically used to diagnose problems in z/OS production environments.

The tools in this Redpaper provide you with the functionality to diagnose problems that affect your important production workload, but are only re-creatable in high transaction environments.

We see these approaches being driven initially by the systems programming staff because they will have authority to access the SVC Dumps or Transaction Dumps taken during failures. Systems programming staff also have the authority to request console dumps of hung or looping servers. Once the unformatted dumps are available, the post-processing and interpretation of the data can be done by either systems programming or development staff, because the tools we document here are Java-based and not tied to z/OS.

Svcdump.jar

This section describes com.ibm.jvm.svcdump.Dump, com.ibm.jvm.findroots, and the API. All are shipped in the svcdump.jar file. We cover installation and usage, as well as arguments and properties that modify processing. We also deliver sample output to enhance your understanding of the scope of these tools.

What is svcdump.jar?

Svcdump.jar enables access directly to the binary SVC Dump or Transaction Dumps created on z/OS, without the need for intermediate software such as the Interactive Problem Control System (IPCS).

There are several utilities shipped in svcdump.jar:

- Dump utility

  The com.ibm.jvm.svcdump.Dump package formats native and Java stacks for threads in dumped processes that include an instantiated JVM. This utility includes functions to print useful information, such as core trace buffers maintained by the JVM and the system trace. These buffers mimic or extend the information that can be obtained with IPCS.

- FindRoots utility

  The com.ibm.jvm.findroots.* package provides multiple ways of formatting the object graphs present in the Java managed heap. This is critical for the sometimes difficult tasks of finding object leaks and, in general, making sense of heap occupancy.

- Java API

  The Java API can be used to write adhoc utilities. For example, you can write small programs to report Java heap objects that maintain state data about a business application.

How to get svcdump.jar

Download the svcdump.jar file from the following Website:

`https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=diagjava`

This tool is under active development. We appreciate your feedback.

Note: This paper is based on the 20041012 version of the code. Later versions might offer additional function or different output.

When to use the Dump utility

If your application crashes, you can use the Dump utility to establish:

- On which thread the crash occurred
- The native stack on the failing thread
- The Java stack for the failing thread

If you experience an application loop or hang, then you can use the utility to identify:

- The thread under which a loop is occurring
- The threads contending for resources or involved in a lockout
- A thread waiting for an operation external to the server

In all of these cases, the reports can help you assign the failure to a particular component or subcomponent before you report an incident to the IBM service team. As an example, assume you have a couple of problems deploying an updated version of an application:

- A lockout seems to occur.

  Inbound workload backs up and you console dump the servant address space involved before recycling the J2EE server.

  – Solution: Using the Dump utility, you find that the problem is a lockout between two threads running framework code developed in-house. Knowing this, you can circumvent IBM Service entirely. With the PMR updates and documentation transmission that goes with it, turn the problem directly over to your in-house development staff.
- A crash occurs during a period of heavy load, and a Transaction Dump is taken when the servant terminates.

  – Solution: You discover that a crash has occurred in a JDBC native method. You are able to direct the PMR you raise with IBM Service to the correct IBM product support group in a timely manner, reducing your overall time to resolve the problem.

How to use the Dump utility on z/OS

There are three files you need from the svcdump.jar download:

- svcdump.jar
- doc.jar, containing documentation for the exposed API
- libsvcdump.so - dll

  libsvcdump.so - dll allows the Java code to access an unformatted dump in an MVS™ dataset, rather than in the hierarchical file system (HFS). This avoids the requirement to provide a large HFS dataset to which to copy the dump. You can, instead, analyze the original dump on z/OS.

To use the Dump utility, copy the three files in binary format to a suitable location in the HFS. In our example, the files are in /u/dclarke.

Use the following command to confirm the version of the utility you are running:

```
:java -cp svcdump.jar com.ibm.jvm.svcdump.Dump –version
```

Example 1 shows a sample output from this command:

Example 1   Determining the Dump utility version

```
You are using
jar:file:/C:/Documents%20and%20Settings/Administrator/My%20Documents/SVCDumps/s
vcdump20041007.jar!/com/ibm/jvm/svcdump/Dump.class
which was last modified on Tue Oct 12 14:53:29 BST 2004
```

Example 1 uses introspection to identify when this code was last modified.

Example 2 is a simple shell script that you can use to execute the utility:

Example 2   Shell script for the Dump utility

```
#!/bin/sh
#TZ=EST5EDT
set -x
DUMPNAME=£1
SVCDUMPJARFILE=/u/dclarke/svcdump20041007.jar
SVCDUMPLIBPATH=/u/dclarke

java -Xmx348m -Dsvcdump.libpath=£SVCDUMPLIBPATH
-Xbootclasspath/p:£SVCDUMPJARFILE \
   -Dsvcdump.default.jvm=0 \
   com.ibm.jvm.svcdump.Dump -exception \
   £DUMPNAME \
   >>£DUMPNAME.svcdump.txt

java -Xmx348m -Dsvcdump.libpath=£SVCDUMPLIBPATH
```

```
-Xbootclasspath/p:£SVCDUMPJARFILE \
   -Dsvcdump.default.jvm=0 \
   com.ibm.jvm.svcdump.Dump -hpitrace  \
   £DUMPNAME \
   >>£DUMPNAME.svcdump.txt

java -Xmx348m -Dsvcdump.libpath=£SVCDUMPLIBPATH
-Xbootclasspath/p:£SVCDUMPJARFILE \
   -Dsvcdump.default.jvm=0 \
   com.ibm.jvm.svcdump.Dump -systrace  \
   £DUMPNAME \
   >>£DUMPNAME.svcdump.txt
```

The shell script can then be invoked. See Example 3:

Example 3   invoke shell script

```
/u/dclarke:==>svcdump.sh ONTOP.GS031.P10316.C724.JVMDMP
+ DUMPNAME=ONTOP.GS031.P10316.C724.JVMDMP
+ SVCDUMPJARFILE=/u/dclarke/svcdump20041007.jar
+ SVCDUMPLIBPATH=/u/dclarke
+ java -Xmx348m -Dsvcdump.libpath=/u/dclarke -
Xbootclasspath/p:/u/dclarke/svcdump20041007.jar -
Dsvcdump.default.jvm=0 com.ibm.jvm.svcdump.Dump -exception
ONTOP.GS031.P10316.C724.JVMDMP
+ 1>> ONTOP.GS031.P10316.C724.JVMDMP.svcdump.txt
```

Analysis of the dump can take some time, especially for the first execution. The tool stores some heap information in a small .cache file, making subsequent executions faster. Use this simple job control language (JCL) to run the utility in a batch:

```
//STEP1    EXEC PGM=BPXBATCH,REGION=0M,
// PARM='SH /u/dclarke/svcdump.sh ONTOP.GS031.P10316.C724.JVMDMP'
```

The Dump utility default report

The default report is created when you run the Dump utility with the default settings. It is typically the first report you run. It tells you if JVMs can be found in any dumped processes. If they are, it dumps out the following for each process:

- Dump title and time
- Full version string for the JVM
- Native and Java stacks for the threads in the process
- Loaded dlls

- Environment variable values
- LE (Language Environment®) runtime options (RTO) values for HEAP and STACK
- Entries from the systrace for each thread, resolved to a code function in LE dlls

The extract in Example 4 shows the report for one thread in one process, both the native and Java stack for the same thread. It also shows the environment variables and loaded dlls for each process for each address space found in the dump. In this case, a thread is waiting on a socket for a reply from a Lightweight Directory Access Protocol (LDAP) server.

Example 4 Dump utility default report (truncated)

```
TCB 9c3288 tid 262718c0 pthread id 254542100000000c tid type 0x00000000 tid state 0x00000017
tid singled 0x00000002 jvmp 268a2000 ee 262716a0 caa 3c0ce560 (non-xplink)
Dsa        Entry      Offset     Function         Method     Module
---        -----      ------     --------         ------     ------
3c0d33b0   091d01e0   f6e2fe9a   CEEOPCT    (pthread_cond_timedwait)
3c0d32f0   09069398   00000084   pthread_cond_timedwait
3c0d31f8   7c9018e0   000002ae   condTimedWait /u/sovbld/cm131s/cm131s-20040117/src/hpi/pfm/condvar_md.c
3c0d30f0   7c913278   00000504   sysMonitorWait /u/sovbld/cm131s/cm131s-20040117/src/hpi/pfm/monitor_md.c
3c0d3028   7cc167c0   000001fc   lkMonitorWait /u/sovbld/cm131s/cm131s-20040117/src/jvm/sov/lk/monitor.c
3c0d2f68   7cae1960   000001b6   JVM_MonitorWait /u/sovbld/cm131s/cm131s-20040117/src/jvm/sov/ci/jvm.c
3c0d2e88   27644994   000000de   java/lang/Object.wait(J)V java/lang/Object.java
3c0d2d90   7cd2e430   00000534   mmipSelectInvokeJavaMethod com/sun/jndi/ldap/Connection.readReply
3c0d2c80   7cd2e518   0000044c   mmipSelectInvokeSynchronizedJavaMethod
   com/sun/jndi/ldap/LdapClient.ldapBind
3c0d2b48   7cd2e518   0000044c   mmipSelectInvokeSynchronizedJavaMethod
com/sun/jndi/ldap/LdapClient.authenticate
3c0d2a20   7cd2e430   00000534   mmipSelectInvokeJavaMethod com/sun/jndi/ldap/LdapCtx.connect
3c0d2928   7cd2e430   00000534   mmipSelectInvokeJavaMethod com/sun/jndi/ldap/LdapCtx.<init>
3c0d2820   7cd2e430   00000534   mmipSelectInvokeJavaMethod
   com/sun/jndi/ldap/LdapCtxFactory.getInitialContext
<..>
3c0cf700   244622d0   00000242   SR_ExecutionThread::RemoveAndProcessWork(ThreadCleanUp*,TCB*)
3c0cf630   24445a70   00000108   SR_ExecutionRoutine
3c0cf578   00000000   09268b2a   (unknown)

Java stack:

Method                                                                   Location
------                                                                   --------
java/lang/Object.wait                                                    Native Method
com/sun/jndi/ldap/Connection.readReply                                   Connection.java:311
com/sun/jndi/ldap/LdapClient.ldapBind                                    LdapClient.java:329
com/sun/jndi/ldap/LdapClient.authenticate                                LdapClient.java:160
com/sun/jndi/ldap/LdapCtx.connect                                        LdapCtx.java:2405
com/sun/jndi/ldap/LdapCtx.<init>                                         LdapCtx.java:258
com/sun/jndi/ldap/LdapCtxFactory.getInitialContext                       LdapCtxFactory.java:91
javax/naming/spi/NamingManager.getInitialContext                         NamingManager.java:674
javax/naming/InitialContext.getDefaultInitCtx                            InitialContext.java:255
```

```
javax/naming/InitialContext.init                                     InitialContext.java:231
javax/naming/InitialContext.<init>                                   InitialContext.java:207
javax/naming/directory/InitialDirContext.<init>                      InitialDirContext.java:92
com/ibm/ws/naming/ldap/WsnLdapInitCtxFactory.getLdapInitialContext   WsnLdapInitCtxFactory.java:255
com/ibm/ws/naming/ldap/WsnLdapInitCtxFactory.getInitialContext       WsnLdapInitCtxFactory.java:171
com/ibm/ws/naming/util/WsnInitCtxFactory.getLdapRootContext          WsnInitCtxFactory.java:436
com/ibm/ws/naming/util/WsnInitCtxFactory.getRootJndiContext          WsnInitCtxFactory.java:322

<..>
com/ibm/ws390/wc/container/WebContainer.init                         WebContainer.java:61
com/ibm/ws390/WebContainerHook.init                                  WebContainerHook.java:89
com/ibm/ws390/rmi/corba/ORBEJSBridge.initOnce                        ORBEJSBridge.java:1090
com/ibm/ws390/rmi/corba/ORBEJSBridge.threadInit                      ORBEJSBridge.java:1068
```

Properties that alter Dump utility behavior

You can set various properties to control svcdump (or jformat). These properties can have default values specified in a properties file called .svcdumprc in your home directory, or you can specify them on the command line as system properties with -D. The -D is a an option for the Java command, not the Print command. Logically, the -D must appear before the Print classname, in one line. See Example 5.

Example 5 The -D property

```
java -Dsvcdump.default.asid=ae -classpath svcdump.jar
     com.ibm.jvm.svcdump.Dump filename
```

Table 1 shows dump properties and their default values.

Table 1 Dump properties

Property	Default	Meaning
svcdump.default.asid	none	Mainly for use with jformat, this property specifies which address space id (asid) to use as the default. The value must be given in hex
svcdump.default.jvm	none	In the case where there is more than one JVM present in an address space, this property specifies the index of the default JVM to use.
svcdump.heapbase	none	This property specifies the heapbase to use when running with options such as -verifyheap. This is useful when the heap corruption prevents the heap walk from completing.
svcdump.numargs	4	This property specifies how many arguments to print when the -args flag is used.
Svcdump.deref.args	false	This property prints what each argument points to (if possible)

Parameters and options to set up the Dump utility

Table 2 provides an overview of the options for the Dump utility. To use the Dump utility with these options, use the following command:

```
java com.ibm.jvm.svcdump.Dump [options] <filename>
```

Table 2   Options for the Dump utility

Option	Description
-debug	print internal debugging information
-verbose	print extra information
-heap	print a table showing which classes have the most objects allocated
-cache	print alloc cache
-exception	print old exception objects
-dis <addr> <n>	disassemble <n> instructions starting at <addr> (hex)
-dump <addr> <n>	dump <n> words of storage starting at <addr> (hex)
-dumpapars	print the apars installed
-dumpclasses	print info about all classes
-dumpclass <addr>	print info about class at given address
-dumpobject <addr>	print info about object at given address
-dumpmdata <addr>	print info about mdata at given address
-dumpprops	print the system properties
-dumpnative	dump all the native methods
-dumpverbosegc	dump the verbosegc
-heapstats	print stats about heap usage
-tcbsummary	print a summary of what the tcbs are doing
-systrace	print the system trace
-hpitrace	print the hpi trace
-caa <addr>	specify the caa to use when disassembling
-r<n>	include saved register <n> in stack trace
-args	print first four function arguments

Option	Description
-verifysubpools	verify subpools
-verifyheap	verify heap
-printdosed	print pinned and dosed objects
-printroots	print GC roots
-version	print the version and exit
-fullversion	print the version of the jvm in the dump and exit
-title	print title of the dump and exit
-time	print time of the dump and exit-dis <addr> <n> disassemble <n> instructions starting at <addr> (hex)

Some parameter samples which we find particularly useful for problem determination in WebSphere for z/OS are described in more detail in the following sections.

Note: In the following examples, <. . .> indicates that we truncated output to show only the important information.

-heap

The -heap option prints a table showing which classes have the most objects allocated.

Example 6 -heap

```
*** dump of live objects in the heap ***

count   class
-----   -----
71866   java/lang/String
70920   array of char
12164   java/util/HashMap$Entry
11008   java/lang/StringBuffer
6560    java/util/Hashtable$Entry
5474    java/util/zip/ZipEntry
<..>
```

```
1      class
com/ibm/ejs/models/base/extensions/webappext/gen/impl/JSPAttributeGenImpl$JSPAt
tribute_List

end heap analysis, cache misses = 8962
total objects = 226518 total length = 18516232
```

In Example 6, the heap has 226518 objects occupying some 17.67 MB.

-alloc

The -alloc option prints the alloc cache. One of the optimizations in the storage subcomponent of the JVM is the alloc cache or Thread Local Heap (TLH). This is maintained on an individual thread basis and provides an area of storage in which small objects can be allocated without the overhead involved in allocating the object in the normal heap. It is sometimes useful to be able to see objects recently allocated in the TLH by a thread. Example 7 shows an empty alloc cache.

Example 7 Empty alloc cache

```
alloc cache info:
cache_busy = 0x0
cache_block = 0x0
cache_size = 0x0 -> 0x0
cache_orig_size = 0x0 -> 0x0
target_size = 0x0
```

Example 8 shows a sample of an alloc cache with entries:

Example 8 alloc cache with entries

```
alloc cache info:
cache_busy = 0x0
cache_block = 0x2bba8340
cache_size = 0x7ac -> 0x2bba8aec
cache_orig_size = 0x20e4 -> 0x2bbaa424
target_size = 0x20000
cache link size = 0x20f0
cache obj size = 0x20e4
cache flags = 0x42

address    length   methods    flags      class name
-------    ------   -------    -----      ----------
2bba8aec   20       3          12         array of java/lang/Object
2bba8b0c   20       250856a0   0          java/util/Vector
2bba8b2c   20       255806a0   80000600   com/sun/jndi/ldap/LdapRequest
2bba8b4c   20       250823a0   0          java/lang/String
2bba8b6c   48       1d         2a         array of char
```

```
2bba8bb4   28       17         42         array of byte
2bba8bdc   58       45         42         array of byte
2bba8c34   20       250823a0   0          java/lang/String
2bba8c54   18       2518afc0   0          java/lang/String$4
<..>
```

-exception

The -exception option prints an old exception object associated with a thread. This can be useful because of the nature of Java's try catch handling of exceptions. This can mask an underlying exception, as in Example 9:

Example 9 -exception option output

```
found old exception: java/lang/ClassNotFoundException:
com.sun.jndi.ldap.LdapCtxFactory
or
found old exception: java/net/UnknownHostException: IZX5
```

-dis <addr> <n>

The -dis <addr> <n> option disassembles n instructions from hex address addr, as in Example 10.

Example 10 -dis parameter output

```
Disassembly starting at 0x024fb388
0x024fb388: (0x00000000): L      $r4, x'0'($r1)
0x024fb38c: (0x00000004): CLC    x'0'(8,$r4),x'e4'($r12)
0x024fb392: (0x0000000a): BNE    x'5e'($r12)
0x024fb396: (0x0000000e): ICM    $r8,15,x'8'($r4)
0x024fb39a: (0x00000012): BEQ    x'5e'($r12)
0x024fb39e: (0x00000016): ICM    $r9,15,x'c'($r4)
0x024fb3a2: (0x0000001a): BNE    x'62'($r12)
0x024fb3a6: (0x0000001e): SLR    $r15, $r15
```

-dump <addr> <n>

The -dump <addr> <n> option dumps n words of storage starting at hex address addr, as in Example 11.

Example 11 -dump parameter output

```
Storage dump starting at 0x024fb388
0x024fb388: 0x58401000
0x024fb38c: 0xd5074000
0x024fb390: 0xc0e44770
0x024fb394: 0xc05ebf8f
0x024fb398: 0x40084780
```

```
0x024fb39c: 0xc05ebf9f
0x024fb3a0: 0x400c4770
0x024fb3a4: 0xc0621fff
```

-dumpclasses

The -dumpclasses option prints all methods with their unique signatures and all static fields within the class, as in Example 12. This can be useful if you suspect there is a mismatch between an updated class library and native code following service application. For example, –dumpclass addr can be used to print just one class.

Example 12 -dumpclasses parameter output

```
Dump of all classes in asid 1f7
Scanning heap, please wait...

*** dump of classes in the heap ***
class java/net/PlainSocketImpl (26a23618) extends java/net/SocketImpl
(26a23718)
attributes: 6140
shared_class_id: 0
initial_static_variable_values: 0
methods:
    method <init> ()V
    method create (Z)V
    method connect (Ljava/lang/String;I)V
    method connect (Ljava/net/InetAddress;I)V
    method connectToAddress (Ljava/net/InetAddress;I)V
    method setOption (ILjava/lang/Object;)V
    method getOption (I)Ljava/lang/Object;
    <..>
    SocketImpl;)V
    method socketAvailable ()I
    method socketClose ()V
    method socketShutdown (I)V
    method initProto ()V
    method socketSetOption (IZLjava/lang/Object;)V
    method socketGetOption (I)I
    method <clinit> ()V
fields:
    static field preferredConnectionTimeout I
    field timeout I
    field closeLock Ljava/lang/Object;
    static final field SOCKS_PROTO_VERS I
    static final field SOCKS_REPLY_VERS I
    static final field COMMAND_CONNECT I
    static final field COMMAND_BIND I
    static final field REQUEST_GRANTED I
    static final field REQUEST_REJECTED I
    static final field REQUEST_REJECTED_NO_IDENTD I
    static final field REQUEST_REJECTED_DIFF_IDENTS I
    static final field socksServerProp Ljava/lang/String; (value = 0x2bba6018)
    static final field socksPortProp Ljava/lang/String; (value = 0x2bba5fd0)
    static final field socksDefaultPortStr Ljava/lang/String; (value = 
0x2bba5f98)
    field shut_rd Z
    field shut_wr Z
    field socketInputStream Ljava/net/SocketInputStream;
    static final field SHUT_RD I
    static final field SHUT_WR I
<..>
```

-dumpobject <addr>

The -dumpobject <addr> option prints information about the object at a particular address:

```
0x2bba5fd0: java/lang/String link bits 20 is live true is marked false
```

This indicates the type of the object, and whether the alloc bit is set (`is live true`) for this chunk of the heap, or whether this chunk of the heap has had the mark bit set during garbage collection (`is marked true`).

-dumpmdata <addr>

The -dumpdate <addr> option prints information from the method block at this particular address. The method block maintains information about a specific method, such as:

```
Dump of mdata at 0x281c6f4c
Method: java/io/BufferedInputStream.read(.BII)I
```

-dumpproperties

The -dumpproperties option prints property values that were set with the –D arguments that passed to the JVM. It can act as a useful check. For example, you can check whether or not changes made in the WebSphere Administrative Console have been applied to the correct server. See Example 13.

Example 13 -dumpproperties output

```
Dump of system properties in asid 1f7
Scanning heap, please wait...

*** dump of system properties ***
```

```
key     value
---     -----
com.ibm.ws390.wc.config.filename
/WebSphere/KWA/controlinfo/envfile/BE01CF1/KWA1AS2C/webcontainer.conf
db2oma2    pbqopc
sun.io.unicode.encoding  UnicodeBig
java.compiler                                           jitc
<..>
```

-dumpnative

The -dumpnative option prints all the native methods in a dump, as in Example 14.

Example 14 -dumpnative output

```
*** dump of all native methods in the heap ***

java/util/TimeZone.getSystemTimeZoneID
(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
java/io/ObjectInputStream.latestUserDefinedLoader ()Ljava/lang/ClassLoader;
java/io/ObjectInputStream.bytesToFloats ([BI[FII)V
<..>
```

-dumpverbosegc

For recent levels of JDK Service, the JVM maintains an in-storage version of the data. This data is printed when you set the –verbose:gc switch. The –dumpverbosegc option prints this buffer. In Example 15, one of its uses is to see if garbage collection pause times are increasing.

Example 15 -dempverbose gc output

```
af   size   free        total       mark   sweep   compact   elapsed
--   ----   ----        -----       ----   -----   -------   -------
1    524    263511000   268368384   16     4       0         20
```

Table 3 provides explinations for each column.

Table 3 Explinations of -dempverbose gc columns

Key	Explination
af	Allocation failure counter, incremented each time an allocation failure causes entry to a garbage collection cycle
size	Size in bytes of the array or object being allocated which led to starting the cycle

Key	Explination
free	Free space on the Java-managed heap after the cycle has ended
total	Total heap size
mark, sweep, compact	Time in milliseconds for these parts of the garbage collection cycle to complete
elapsed	Sum of mark, sweep and compact times, in milliseconds

-heapstats

The -heapstats option prints summary information about the Java heap, as in Example 16.

Example 16 -heapstats output

```
Print heap stats in asid 1f7 jvmp 0x268a2000
total objects = 139669
total classes = 5299
total objectarrays = 7801
total primarrays = 76812
total strings = 71866
total smallid = 113822
total mediumid = 25835
total largeid = 12
total smallrefs = 97375
total mediumrefs = 11885
total largerefs = 24634
total smallsize = 69189
total mediumsize = 7609
total largesize = 14
total refs[0] = 4741
total refs[1] = 102713
total refs[2] = 19105
total refs[3] = 7094
total refs[4] = 2285
total refs[5] = 2390
total refs[6] = 307
total refs[7] = 1034
total cachehits = 40893
total cachemisses = 26910
total fithits = 111834
total fitmisses = 27835
total oldsize = 2420174
total newsize = 1710807
total hashed = 168
```

-tcbsummary

The -tcbsummary collates cases where threads have identical native stacks, as in Example 17. This is useful if a process has many threads, and the majority of them are in typical waits for work status. Object waits or other waits are unlikely to be of interest from a diagnostic perspective.

Example 17 -tcbsummary output

```
<..>
The following tcbs had this traceback:
9cc1e8   9c30f0   9c1190   9bf288   9bd288   9bb288   9bb0f0   9b9288
9b7288   9b70f0   9b5190

    CEEOPCW
    pthread_cond_wait
    condWait
    sysMonitorWait
    lkMonitorEnter
    @@GETFN
    mmipSelectInvokeJavaMethod
    INVOKDMY
    EXECJAVA
    mmipExecuteJava
    xeRunJniMethod
    jni_CallStaticVoidMethodA
    ORBEJSBridge::threadInit()
    SR_ExecutionThread::RemoveAndProcessWork(ThreadCleanUp*,TCB*)
    SR_ExecutionRoutine
    (unknown)
<..>
```

-systrace

The -systrace option prints a version of the z/OS system trace, formatted by asid and time. For Language Environment functions, it will resolve the program status word (PSW) from the interrupt to a function. It is useful for identifying a thread running garbage collection code during a garbage collection cycle. See Example 18.

Example 18 -systrace output

```
*** Trace for asid 6d started at Fri Mar 05 16:04:31 GMT 2004 ***

Time          Tcb       Psw        Offset     Type   Function
----          ---       ---        ------     ----   --------
16:04:31:749  1b48088   80fe6278   0          SRB    unknown function
16:04:31:749  1b8bdc0   80fe6278   fe6278     SRB    unknown function
16:04:31:749  adfe88    fc477af4   660        DSP
is_inlinable_method_invocation
```

```
16:04:31:750 adfe88 fcd6ff26 20   I/O xmIsJVMResettable
16:04:31:750 adfe88 868a9abc 19d0  SVC  CEEOPCW
16:04:31:750 adfe88 868a9abc 19d0  SVCR CEEOPCW
16:04:31:750 adf798 868a9abc 19d0  DSP  CEEOPCW
16:04:31:755 adf798 fc45211c 250  CLKC analyze_reachable_bb
16:04:31:757 adf798 fc45211c 250  DSP analyze_reachable_bb
16:04:31:758 adf798 fc943bf0 bc   EXT  SYSNCPU
16:04:31:759 adf798 fc3f258e b0  EXT search_override_method
16:04:31:759 adf798 fcb723c6 90   EXT clAddUTF8String
16:04:31:759   adf508    813698d6   0         DSP    unknown function
<..>
```

-hpitrace

The -hpitrace option accesses the HPI subcomponent of the JVM. This subcomponent is the layer closest to the underlying operating system. It maintains an in-storage buffer tracing recent calls. Printing this buffer can help identify otherwise hard to spot problems resulting from failing operations such as pthread_quiesce_and_get_np() that are used to freeze threads prior to a garbage collection cycle. See Example 19.

Example 19   -hpitrace output

```
found hpi trace, offset = 00001dcc table start = 26931dc0 table end = 269b1dc0 table size =
00080000 asid = 000001f7
Trace begins at Fri Oct 08 09:24:40 BST 2004
09:24:40:475 tcb 009f91c8 Bootstrap: po_id(x%08X), threadself(x%08X%08X) 261fd460 25411040
00000000
09:24:40:671 tcb 009f91c8 sysFileType: (%s), file does not exist (or path is NULL)
09:24:40:804 tcb 009c9d90 Shell: tid(x%08X), threadself(x%08X%08X) 2625b188 25413710 00000002
09:24:40:807 tcb 009c9a78 Shell: tid(x%08X), threadself(x%08X%08X) 2625b550 25414400 00000003
09:24:40:811 tcb 009c9760 Shell: tid(x%08X), threadself(x%08X%08X) 2625b918 25416ad0 00000004
09:24:40:813 tcb 009c9448 Shell: tid(x%08X), threadself(x%08X%08X) 2625bce0 254177c0 00000005
09:24:40:815 tcb 009e0cc8 Shell: tid(x%08X), threadself(x%08X%08X) 2625c0a8 254184b0 00000006
09:24:40:818 tcb 009e09b0 Shell: tid(x%08X), threadself(x%08X%08X) 2625c470 2541b870 00000007
09:24:42:932 tcb 009f91c8 sysOpen: fd(%d), path name is not valid, errno(%d) ffffffff 00000081
09:24:51:570 tcb 009e00b0 Shell: tid(x%08X), threadself(x%08X%08X) 2626d878 2541df40 00000008
09:24:53:450 tcb 009ccbe0 sysThreadAlloc: tid(x%08X), threadself(x%08X%08X) 26271130 25450160
0000000a
09:24:53:455 tcb 009cc950 sysThreadAlloc: tid(x%08X), threadself(x%08X%08X) 262714f8 25438a30
00000009
09:24:53:458 tcb 009c3288 sysThreadAlloc: tid(x%08X), threadself(x%08X%08X) 262718c0 25454210
0000000c
09:24:53:462 tcb 009b70f0 sysThreadAlloc: tid(x%08X), threadself(x%08X%08X) 26271c88 25461110
00000016
09:24:53:465 tcb 009bd288 sysThreadAlloc: tid(x%08X), threadself(x%08X%08X) 26272050 254575d0
0000000e
```

-args

A combination of the –args option and property -Dsvcdump.deref.args=true can be used to make more sense of the native stacks printed for each thread as in Example 20:

Example 20 -args sample output

```
TCB 9e64e8 tid 00000000 jvmp 00000000 ee 00000000 caa 24f27558 (non-xplink scb)
Dsa      Entry    Offset     Function     Module
---      -----    ------     --------     ------
24f313b8 08e25348 1c10bfae read(15(->7f), 29fd8d90(->ee8), 1000(->0), 24f31418(->1084e1b0))
24f312e8 25d2d288 00000130 getdata(27a6fcc8(->15), 1(->a000000), 27a56ff0(->2538c950), 88df1a24(->?))
24f31218 25d2dd10 000000da rgets(24f2b160(->258a07d0), 1ffe(->0), 27a6fcc8(->15), 2(->1))
24f31160 25cf6ab0 00000098 readLine(253936b0(->15), 24f2b160(->258a07d0), 1fff(->0), 25d407e0(->c1c2c3c4))
24f2b080 25d29748 000000e2 htresponseRead(29ff3110(->29ff3548), 253936b0(->15), 0(->?), 0(->?))
24f2afa8 25d00bd8 00000506 websphereExecute(24f2af18(->24f28f78), 24f2af38(->0), 25392e68(->29ff2c50),
25d46860(->c9d5c9e3))
24f2ae70 25d078d8 000005aa websphereHandleRequest(24f28f78(->24f28fe4), 25d4697c(->d8e4c5d9), c(->0), 24f2aa50(->0))
24f28ec8 25d28300 000008be service_exit(70021(->?), 24f28e2c(->0), 25514d48(->6188a394), 0(->?))
24f28d70 256d5538 00000d68 ApiClassExecute(258a3238(->a28599a5), 25249ca8(->c9d4e6d3), 258a32d8(->60a49592)
24f28d34(->258a3974))
24f28c58 257b55a0 0000048e RedirectionOntheFly(25249e38(->0), 25249ca8(->c9d4e6d3), 25249ca8(->c9d4e6d3),
25249ca8(->c9d4e6d3))
24f28b68 257b87a8 00000c84 HTHandle(25249ca8(->c9d4e6d3), 2(->1), 25249ca8(->c9d4e6d3), 24f28868(->8))
24f28970 257b72b8 000012cc HTSession(24f288c0(->12), 25249ca8(->c9d4e6d3), 8(->0), 24f28884(->1))
24f287c8 257588d8 00000c70 server_loop(25509fc8(->252149a8), 25249ca8(->c9d4e6d3), 0(->?), 0(->?))
24f28628 256b69e0 00000762 HotThread(25509fbc(->252873b0), 24f28530(->24f28540), 0(->?), 0(->?))
24f28570 00000000 09268b2a (unknown)
```

-verifyheap

You can use the -verifyheap option to look for errors in the heap. You might use this option following an unexpected crash in the interpreter (functions starting mmi*) or a just-in-time (JIT) method. See Example 21.

Example 21 -verifyheap output

```
Verifying heap in asid 1f7 jvmp 0x268a2000
analyzing classes
analyzing middleware heap from 2a8801fc to 3a86fbfc
analyzing heap at 2a8801fc total objects so far = 1
analyzing heap at 2b2f6064 total objects so far = 109021
analyzing transient heap from 0 to 0
analyzing thread local heap for tcb 9c9760 from 2a9aea74 to 2a9aec24
analyzing thread local heap for tcb 9c3288 from 2bba8aec to 2bbaa424
analyzing thread local heap for tcb 9bf028 from 2bbab46c to 2bbab4ec
```

-verifysubpools

Use the -verifysubpools option to enable looking for errors in the JVM structures that manage the JVM use of native memory. It is unrelated to the z/OS VSM component.

-printdosed

The -printdosed option identifies dosed objects. These are objects considered to be roots for the marking phase of garbage collection. They are set dosed to make sure they cannot be moved during a later compaction phase. See Example 22.

Example 22 -printdosed output

```
Print dosed/pinned in asid 1f7 jvmp 0x268a2000
found dosed/pinned bits 4 in object 2a98f540 class
com/ibm/ws/classloader/SinglePathClassProvider
found dosed/pinned bits 4 in object 2a97dcb0 class org/w3c/dom/Entity
found dosed/pinned bits 4 in object 2a982928 class
org/apache/xerces/validators/common/XMLValidator$ValueStoreCache
found dosed/pinned bits 4 in object 2a916368 class
org/apache/xerces/validators/common/CMLeaf
found dosed/pinned bits 4 in object 2a913068 class
com/ibm/etools/j2ee/common/meta/impl/MetaSecurityRoleImpl
found dosed/pinned bits 4 in object 2a91e2e0 class
com/ibm/etools/commonarchive/meta/impl/MetaModuleComponentImpl
found dosed/pinned bits 4 in object 2a8ce810 class
com/ibm/websphere/ras/Manager
found dosed/pinned bits 4 in object 2a8cb510 class org/omg/CORBA/BOA
<..>
```

-printroots

The -printroots option prints the root objects. The FindRoots utility does a more useful job of this type of analysis from the perspective of diagnosing object leaks. See Example 23.

Example 23 -printroots output

```
Print GC roots in asid 1f7 jvmp 0x268a2000
mh base = 2a8801fc limit = 3a86fbfc
scanning heap for valid objects
scanning tcb stacks
analyzing classes
found stuck object: 269c0118 reason: nativeobject class java/lang/Class
found stuck object: 269c0218 reason: nativeobject class java/lang/Object
found stuck object: 269c0518 reason: nativeobject class java/lang/String
found stuck object: 269c5d18 reason: javaobject class java/lang/System
found stuck object: 2a880200 reason: pinned array of byte
found stuck object: 2a880210 reason: pinned class
org/apache/xerces/dom/CDATASectionImpl
<..>
```

When to use the FindRoots utility

FindRoots is a cross-platform tool for analyzing memory leaks in Java applications. It provides multiple ways of formatting the object graphs present in the Java-managed heap. This is critical for the sometimes difficult tasks of finding object leaks and making sense of heap occupancy.

A typical memory leak in Java is caused by an application or middleware bug retaining a reference to an object. Because this object and all of its referents are still reachable, a large portion of an object graph is still reachable. This means it is not eligible for garbage collection.

As an example, imagine some state data for a transaction is kept as an XML data object. The design is such that the data is only relevant for the life of the transaction. The object is eligible for garbage collection after the transaction has ended. However, there is a bug in the code. This bug causes some other global object to maintain a reference to the XML data object after the end of the transaction. Although the object itself is small, it contains a reference to non-trivial numbers of objects created by XML parsing.

A successful problem diagnosis looks like the following:

1. A still reachable XML data object exists after each transaction runs. The reachable data remaining gradually increases over time after each garbage collection cycle.You can observe this increase from the verbose:gc output, or from the incore verbosegc data.
2. Eventually, the Java heap is exhausted and an OutOfMemoryError is thrown. Run a console dump of the server when the heap usage is high.
3. Run PrintDomTree and establish from the reports that there is an unexpectedly high number of these XML data objects.
4. Find the unexpected reference in the reports from the global object.
5. Review the logic and see that this reference is not nulled out after the transaction, as the design anticipated.

How to use the FindRoots utility on z/OS

The FindRoots utility is the com.ibm.jvm.findroots.* package in the svcdump.jar.

Although the package is called FindRoots for historical reasons, it is now split into a number of smaller tools that do the same thing. The split is mainly to separate the extraction of the heap from the subsequent analysis.

Figure 1 on page 22 shows the general approach to use the FindRoots utility.

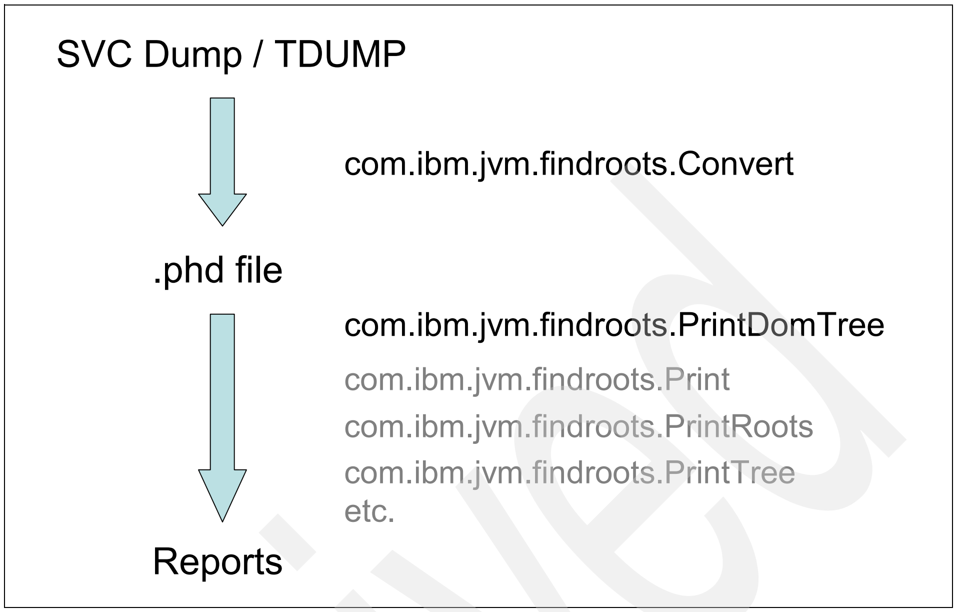

SVC Dump / TDUMP
com.ibm.jvm.findroots.Convert
.phd file
com.ibm.jvm.findroots.PrintDomTree
com.ibm.jvm.findroots.Print
com.ibm.jvm.findroots.PrintRoots
com.ibm.jvm.findroots.PrintTree
etc.
Reports

Figure 1 FindRoots approach for analysis

To use the FindRoots utility, do the following:

1. Run the `Convert` tool to extract the heap dump and create a Portable Heap Dump - a .phd file.
2. Run another tool to analyze the .phd file. Use `PrintDomTree` as a primary solution before you use the others, as in Figure 1.
3. For further detail and analysis, use the other tools:
   - com.ibm.jvm.findroots.Print
   - com.ibm.jvm.findroots.PrintRoots
   - com.ibm.jvm.findroots.PrintTree

The advantage of this approach is that you can discuss and pass the smaller .phd files among your team, rather than large SVC or transaction dumps.

The Convert tool

This tool creates a .phd file.To run the Convert tool, use this command:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.Convert <filename>
```

In some situations, there can be multiple copies of the JVM in the dump. In this case, you see an error message asking you to specify which JVM to use. Specify the index of the JVM by running this command:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.Convert -jvm <index>
     <filename>
```

Note: By default, the Convert tool includes all subclasses of java/lang/ref/Reference from the resulting .phd file. Because they represent paths to objects that should not exist, sometimes it is best not to have all subclasses. They can interfere with FindRoots. These subclasses are treated specially by the Java Garbage Collector. They are, in effect, ignored when it calculates reachability. If you want to exclude references to all subclasses in the .phd, then you must specify the property:

```
-Dfindroots.include.references=false
```

Also note that Finalizer is a reference subclass. When you do this, you will no longer see long chains of Finalizer objects.

When run with the new format .phd files, all of the Print commands also obey:

```
-Dfindroots.include.references=false
```

It is more flexible to keep the references in the .phd file and only exclude them when running the analysis.

Print

Print simply prints the contents of a .phd file. To use the Print tool, execute the following:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.Print <filename>
```

By combining this with traditional Unix filters such as `sort` and `uniq`, you can calculate which class has the most objects, as in the following command:

```
cut -d ' ' -f 3 output | sort | uniq -c | sort -n -r
```

This statement defines and executes the following:

- Removes the third field, using space as a field separator
- Pipes it into sort
- Pipes that into uniq, used to count how many of each repeated line
- Pipes that output again into sort to do a reverse sort based on the count.

Note: The above trick has been supplanted now by the summary information printed at the end of the output. If you want to see only the summary information, then you can specify the option `-summary.` If you only want to see the total heap size and number of objects, specify the `-totals` option.

PrintRoots

PrintRoots does traditional FindRoots analysis on a .phd file. To use PrintRoots, type the following:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.PrintRoots <filename>
```

You can also specify the maximum depth with the `-Dfindroots.depth` property. Maybe this should be called the *width* property, because it means the maximum number of levels of indent.

Tip: The default is currently 1000. If you make it too big, you can get very large output.

PrintTree

PrintTree prints the tree below a given object. This approach short-circuits the usual find roots analysis, so you need to already know the id of the object in which you are interested. It is useful for iterative approaches, for example, if you discover that the depth was not big enough and you do not want to start all over again. To use the PrintTree command, do the following:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.PrintTree -root <id>
  <filename>
```

If you specify `-inverse`, then the inverse of the tree is printed. It prints the tree above the given object, starting with the parents, then their parents and so on.

PrintDomTree

PrintDomTree calculates and prints the dominator tree. The dominator tree displays the parent as the next object up in the original tree that dominates all paths to the child. It collapses the tree, only showing the important parts from the perspective of memory leak analysis. That is because a dominator of a given object is responsible for keeping alive the given object.

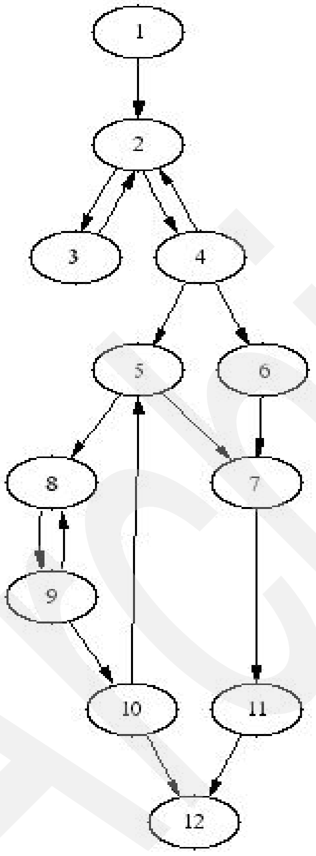
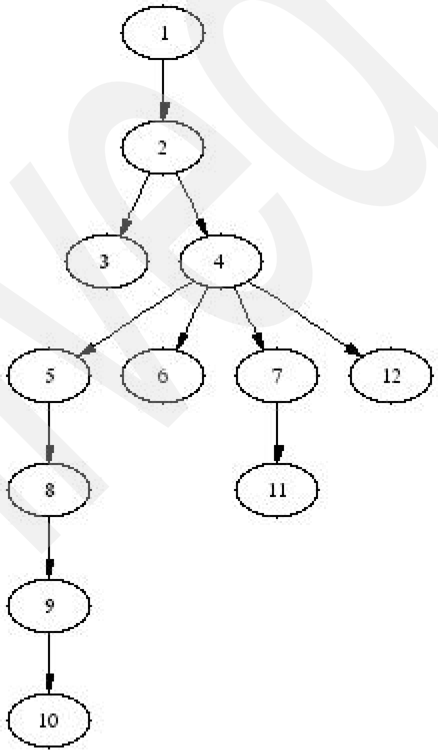
The use of this tool is now our recommended approach to memory leak analysis. To use the PrintDomTree tool, execute the following command:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.PrintDomTree
  <filename>
```

There is also a -root option which can be used to give the hex id of the root of the dominator tree to use. By default, it calculates the object with the greatest reachability and uses it.

Table 4 shows an example of a dominator tree and its original tree.

Table 4 Sample trees

Original tree	**Dominator tree**
	

Notice that the dominator tree on the right does not necessarily reflect the direct relationships between objects. For example, node 7 is a child of node 4 in the dominator tree, but is a grandchild in the original tree on the left.

Tip: The definition of a *strong component* is a group of objects such that every object in the group is reachable from every other. In our example 2, 3 and 4 form one strong component, while 5, 8, 9 and 10 form another.

For further understanding, refer to Example 24, sample output of the PrintDomTree tool.

Example 24 PrintDomTree sample output

```
1877294 0x2b9c0e80 com/ibm/servlet/classloader/DynamicClassLoader
   1255299 0x2b9c1558 com/ibm/servlet/classloader/SystemClassLoader
      1255287 0x2b9c1638 sun/misc/Launcher$AppClassLoader
         5683 0x2b9c16a0 sun/misc/Launcher$ExtClassLoader
            5631 0x2b9f5b68 sun/misc/URLClassPath
               5587 0x2b9f5ac0 java/util/ArrayList
                  5586 0x2ba4e0f8 array of java/lang/Object
                     5585 0x2b9f82b8 sun/misc/URLClassPath$JarLoader
         468 0x2b9ead08 java/util/Hashtable
            467 0x33b89fc8 array of java/util/Hashtable$Entry
         1219496 0x2b9eac78 java/util/Vector
            1219495 0x33d41910 array of java/lang/Object
               1151895 0x2b9709d0 class
com/wily/introscope/agent/blame/ComponentTracer
                  1151892 0x2d25b6d8
com/wily/introscope/agent/stat/DataAccumulatorFactory
                      1150580 0x2d13f4b0
com/wily/introscope/agent/enterprise/EnterpriseAgent
```

The number to the left on each line is the reachability for that object. It defines either the number of objects or the size of memory occupied, depending on whether you set the findroots.sizematters property. The reachability is defined as the number of objects, or size occupied by them, reachable from a given object by following reference links. This number is the total of the children, the children's children and so on.

The second field is the address of the object in the heap. The third field is the class name.

Child objects are indented. In Example 24, DynamicClassLoader has a reference to SystemClassLoader. Therefore, SystemClassLoader is a child and is indented. Similarly, SystemClassLoader points to Launcher$AppClassLoader. Launcher$AppClassLoader has three children: Launcher$ExtClassLoader, Hashtable and Vector.

In order not to clutter the output, objects with only a small reachability are pruned and do not appear. The default for this is 1000 but can be set where applicable with the findroots.prune property. Objects are only printed once. If the same object is pointed to by more than one parent, the object will only appear the first time.

PrintClassReach

The PrintClassReach tool prints the space reachable by all objects of the given class. To use the PrintClassReach to, execute the following command:

```
java -classpath svcdump.jar com.ibm.jvm.findroots.PrintClassReach
  -class <classname> <filename>
```

Use this tool when you suspect that many objects of a given class are causing the problem, rather than one particular object tree. Note that classname should be written in directory style, that is `java/lang/Object`.

PrintComponents

PrintComponents does basically the same thing as PrintRoots, except that it collapses the strong components into just one entry, producing a much less cluttered output. The contents of the strong components are the numbers at the end of the line. See Example 25.

Example 25 PrintComponents sample output

```
30035 0x76ab 0x144a7318 class java/lang/ref/Finalizer
   29983 0x76a9 #30377 51
      29409 0x73df #29663 85
         20164 0x73dd 0x14bbf430 sun/misc/URLClassPath
            20150 0x73dc 0x14bbe160 java/util/ArrayList
               20149 0x73db 0x14bbe288 array of java/lang/Object
                  20135 0x73da #29658 14
                     12238 0x6336 #25398 2
                        12233 0x6335 0x14d33708 java/util/jar/JarVerifier
                           12221 0x6329 0x14ca1b18 java/util/jar/Manifest
                              12211 0x6328 0x14ca1a38 java/util/HashMap
                                 12210 0x6327 0x14d46280 array of java/util/HashMap$Entry
                        12221 0x6329 0x14ca1b18 java/util/jar/Manifest (already visited)
```

Where a strong component has been collapsed, a #nnn is printed, followed by the number of objects in that component. The contents of the component are printed at the end, as in Example 26 on page 28.

Example 26 Collapsed strong component

```
*** #25398 ***
0x14c9d980 java/util/jar/JarFile$JarFileEntry
0x14ca1d10 java/util/jar/JarFile
```

Properties that alter the FindRoots utility behavior

All the Print commands obey a set of standard properties. These properties might have default values specified in a properties file called .svcdumprc in the user's home directory, or they might be specified on the command line as system properties with -D. Note that the -D is a an option for the Java command, not the Print command, so it must appear before the Print classname. For example:

```
java -Dfindroots.depth=2000 -classpath svcdump.jar
  com.ibm.jvm.findroots.PrintRoots filename
```

Table 5 lists all the FindRoots utility properties and their default values.

Table 5 FindRoots properties

Property	Default	Purpose
findroots.depth	1000	This property controls how deep to print levels of indentation in the output tree.
findroots.prune	1000 or 10 000	Objects with reachability smaller than this amount are not printed. This is to avoid cluttering the output with millions of lines for objects that do not contribute much to the whole. The default depends on whether sizematters is used, with the larger value being sizematters=true. For the PrintTree command, the default is zero.
findroots.sizematters	false	This property controls whether to base the reachability on the cumulative size of each object, or the total number of individual objects. In the vast majority of cases, size does not matter. The real issue is the number of objects being kept alive. Switching on sizematters will tell you how much actual space is involved, but it does not change the basic structure of the tree.
findroots.maxroots	20	This property controls how many roots are displayed, where appropriate.
findroots.exact	false	This property controls whether to base the reachability on the cumulative size of each strong component, rather than the much quicker way of treating each component as one element. As with sizematters, turning this on produces much longer running times for the same reason, and in most cases is not necessary. Switched it off effectively treats each strong component as one node.

Property	Default	Purpose
findroots.uselessmemory	false	FindRoots commands can sometimes require a lot of memory to run. When this property is set, it forces the use of a different algorithm to calculate the reachability. This is h is slower but uses less memory. One drawback of this algorithm is that the counts of everything but the root itself are based on a single depth first search of the tree, as a result it is not as accurate.
findroots.include.references	true	When used by Convert, this property controls whether or not to include subclasses of java/lang/ref/Reference in the .phd file. When used by the various Print* tools and set to `false`, it can also be used to exclude instances of such subclasses.
findroots.calculate.reachability	true	For certain commands (for example, PrintTree) this property determines whether or not to calculate the reachability. It takes a lot more memory and time to calculate the reachability. If you want to see the subtree for a particular object only, turn it off.

Tip: TreeViewer is a simple GUI for displaying FindRoots output files. Give it the name of the output file. TreeViewer displays just the top root in a pop-up window. Click the **expand tree** icon to open the next level down and so on. It is much easier to browse big trees like this.

FindRoots utility in practice

This section contains an example of how we analyzed memory leaks using the FindRoots utility.

We used Convert to create the .phd file. The original dump was enormous, 3 GB. However, the resulting .phd file was only 43 MB, which was a lot easier to pass among the team. There are a total of 4.4 million objects in this dump in a heap size of 512 MB.

We ran PrintRoots. The resulting output suggested CachedTargets was central to the leak, but the output was so large it was hard to see details. We then ran PrintDomTree. Example 27 shows the output from PrintDomTree:

Example 27 PrintRoots utility output in practice

```
2547917 0x30a1e8f0 com/ibm/servlet/engine/srt/CachedTargets
   2547916 0x30a331b8 java/util/Vector
      2547915 0x39a15390 array of java/lang/Object
         266 0x3e166068 com/ibm/servlet/engine/srt/WebAppInvoker
            145 0x3e165f88 com/ibm/servlet/engine/srt/SRTServletRequest
               134 0x3e163e80 com/ibm/servlet/engine/srt/http/HttpHeader
```

```
132 0x3e3807f8 array of com/ibm/servlet/engine/srt/http/MimeHeaderField
   3 0x3e23e030 com/ibm/servlet/engine/srt/http/MimeHeaderField
      0 0x3e23e010 com/ibm/servlet/engine/srt/http/MessageString
      0 0x3e23dff0 com/ibm/servlet/engine/srt/http/MessageString
      0 0x3e23dfc8 com/ibm/servlet/engine/srt/http/HttpDate
   3 0x3e23dfa8 com/ibm/servlet/engine/srt/http/MimeHeaderField
      0 0x3e23df88 com/ibm/servlet/engine/srt/http/MessageString
      0 0x3e23df68 com/ibm/servlet/engine/srt/http/MessageString
      0 0x3e23df40 com/ibm/servlet/engine/srt/http/HttpDate
```

Notice the huge difference between the figure of 2547915 objects for the array and only 266 for the WebAppInvoker. We set -Dfindroots.prune=0 to make the children print.

Because children are printed in order of reachability with the biggest first, this implies that the array must have many children. It is most likely that all the children are going to be the same class.

- a grep for WebAppInvoker in the output piped into wc showed that there were 14700 instances of WebAppInvoker, all belonging to the array in CachedTargets.

We found logic that generated a unique URI for every request within the application. This, in turn, skipped the cache and created a new WebAppInvoker object each time. The number of unique URIs for the application appeared to be infinite. Normally an application has, at most, 100 unique URIs.

HeapRoots

The HeapRoots utility is shipped in HR204.jar and derives from the same requirement to be able to map the heap object graphs. HeepRoots was originally developed for the JVM shipped with AIX®. It is now possible to seamlessly use this code with binary SVC or Transaction Dumps, adding a range of functions to that provided with the FindRoots utility in svcdump.jar.

You can use this approach to:

- Confirm the roots analysis done with the FindRoots utility
- Perform a gap analysis for those cases where the allocation of a large object or array fails because of heap fragmentation
- Locate objects of a particular type quickly

How to get the HeapRoots utility

HeapRoots is available from the AlphaWorks Web site:

http://www.alphaworks.ibm.com/tech/heaproots

How to use the HeapRoots utility

To use HeapRoots with .phd files, use the following command:

```
java -classpath svcdump.jar;HR204.jar HR.main.Launcher
```

For illustration purposes, Example 28 shows a session we ran.

Example 28 HeapRoots utility session sample

```
C:\Documents and Settings\Administrator\My Documents\SVCDumps>

java -cp svcdump20041020.jar;HR204.jar HR.main.Launcher ONTOP.GS04
2441.C678.DUMP1.phd

HeapRoots version 2.0.4. Copyright (c) IBM Corporation 1996,2003. All rights reserved.

Opening 'ONTOP.GS040.P52441.C678.DUMP1.phd' as a 'Portable Heap Dump file (typically generated
on z/OS)' ...
Requesting at least 34 mb of heapspace to load heapdump (string data extra) ... done.
.......................................................................
UNEXPECTED DATA: There were 105 unexpected size/addresses. Rerun with verbose option to view
 done.
Sorting objects by address ........................................... done.
Mapping refs from addresses to indexes ......................................... done.
# Unresolved references : 95. Rerun with verbose option to view
```

```
Sorting refs ...
.................................................................. done.
Successfully opened 'ONTOP.GS040.P52441.C678.DUMP1.phd'. This filename was derived from
'ONTOP.GS040.P52441.C678.DUMP1'.

Dump comments        : none
Dump has flags       : false
# Objects            : 910,178
# Refs               : 2,184,181
# Unresolved refs    : 95
Heap usage           : 127,787,448
Total object range   : 135,109,412

Extra stats          : unavailable before processing

Memory Usage         : 23/40 mb

Enter: o{a,s,t,d,m,n}, g{c,s}, t{c,s,n}, i, p, d{t,d,m}, help or x to exit
```

Type `help` for information about the supported commands, and then `help o`, `help g`, and so on, for detailed help about these options. Table 6 shows the supported options:

Table 6 HeapRoots utility options

Option	Function
o/oa/os/ot/od/om/on	Searching and Listing of objects
g/gc/gs	Tabulates the sizes of gaps between objects
t/tc/ts/tn	Tabulates the various names of objects
i {0xaddr}	Shows information about a single object
p/process/k/keep	Processes the dump to find roots and other statistics. This involves searching from each root in the graph. ► Use p 0xaddr to start processing from 0xaddr so that it gets bias when owning objects. ► Use k 0xaddr to process the keep-alive size of 0xaddr. This starts the same as p 0xaddr but then objects reachable by other roots are removed from 0xaddr.
d/dt/dd/dm	Textual interpretation of graph from Roots

In Example 29 on page 33, we used dt to look at the object graph below the com/ibm/servlet/engine/srt/CachedTargets object at 0x1b3da840. This shows that the array is of com/ibm/servlet/engine/srt/WebAppInvoker objects.

Example 29 HeapRoots utility session sample (cont.)

```
> dt

Enter total-size threshold [1048576]
>
Enter max depth or -ve for unlimited [1000]
>
Enter 0x<addr> to dump from one address or any value for all roots [-]
> 0x1b3da840
Dumping object(s)  : 0x1b3da840 (parent 0x1b3f2e50 root 0x1a355e18), sorted by total-size
Filter             : total-size >= 1,048,576 and depth at most 1000
Prune              : N

<0> [24,867,228] 0x1b3da840 [12] com/ibm/servlet/engine/srt/CachedTargets
<1>   [24,867,216] 0x1b3da820 [28] java/util/Vector
<2>     [24,867,188] 0x20cf36f8 [172] array of java/lang/Object
<3>       [4,118,312] 0x20af62e8 [60] com/ibm/servlet/engine/srt/WebAppInvoker
<4>         [4,047,254] 0x20af62c8 [28] java/util/Vector
<5>           [4,047,226] 0x20af6280 [52] array of java/lang/Object
<6>             [4,047,174] 0x20af6620 [44]
      com/ibm/servlet/engine/invocation/CacheableInvocationContext
<7>               [4,046,486] 0x20af6380 [28] java/util/Vector
<8>                 [4,046,458] 0x20af6348 [52] array of java/lang/Object
<9>                   [4,046,406] 0x1a584768 [60]
      com/ibm/servlet/engine/invocation/InvocationCache
<10>                    [4,042,646] 0x1f3e9720 [396] array of java/util/Hashtable$Entry
<11>                      [2,190,556] 0x1e8046a0 [28] java/util/Hashtable$Entry
<12>                        [2,190,352] 0x1de976b8 [44]
      com/ibm/servlet/engine/invocation/CacheableInvocationContext
<13>                          [2,181,704] 0x1de9dcf0 [60]
      com/ibm/servlet/engine/srt/WebAppInvoker
<14>                            [2,134,614] 0x1afdda80 [92]
      com/ibm/servlet/engine/srt/WebGroup
<15>                              [2,008,972] 0x1b052eb8 [68]
      com/ibm/servlet/engine/webapp/WebApp
<16>                                [1,982,566] 0x1b051b40 [44]
      com/ibm/servlet/engine/webapp/WebAppServletManager
<17>                                [1,982,414] 0x1b051490 [44] java/util/Hashtable
<18>                                [1,982,370] 0x1a5f0648 [204]
      array of java/util/Hashtable$Entry
                                    - 19 children of 0x1a5f0648 filtered.
<17>                                {2,008,972} 0x1b052eb8 [68]
      com/ibm/servlet/engine/webapp/WebApp
<17>                                {2,008,972} 0x1b052eb8 [68]
      com/ibm/servlet/engine/webapp/WebApp
                                    - 4 children of 0x1b051b40 filtered.
<16>                                {2,134,614} 0x1afdda80 [92]
      com/ibm/servlet/engine/srt/WebGroup
```

```
                                 - 5 children of 0x1b052eb8 filtered.
<15>                           <25,091,324 parent:0x1b400718> 0x1afc2d18 [28]
      com/ibm/servlet/engine/ServletHost
                               - 12 children of 0x1afdda80 filtered.
<14>                         {2,190,352} 0x1de976b8 [44]
      com/ibm/servlet/engine/invocation/CacheableInvocationContext
                             - 6 children of 0x1de9dcf0 filtered.
<13>                       {2,181,704} 0x1de9dcf0 [60]
      com/ibm/servlet/engine/srt/WebAppInvoker
                           - 5 children of 0x1de976b8 filtered.
                         - 1 child of 0x1e8046a0 filtered.
                       - 33 children of 0x1f3e9720 filtered.
                    - 2 children of 0x1a584768 filtered.
<7>            {4,118,312} 0x20af62e8 [60] com/ibm/servlet/engine/srt/WebAppInvoker
<7>            {4,118,312} 0x20af62e8 [60] com/ibm/servlet/engine/srt/WebAppInvoker
               - 4 children of 0x20af6620 filtered.
<4>         <24,878,758 parent:0x1b420d78> 0x1b3da880 [92] com/ibm/servlet/engine/srt/WebGroup
<4>         {24,867,228} 0x1b3da840 [12] com/ibm/servlet/engine/srt/CachedTargets
<4>         {4,047,174} 0x20af6620 [44]
      com/ibm/servlet/engine/invocation/CacheableInvocationContext
            - 4 children of 0x20af62e8 filtered.
          - 27 children of 0x20cf36f8 filtered.

There were 19 objects expanded.
```

in Example 30, we used the os command to locate the 25 largest items on the heap.

Example 30 HeapRoots utility session sample (cont.)

```
Enter: o{a,s,t,d,m,n}, g{c,s}, t{c,s,n}, i, p, d{t,d,m}, help or x to exit
> os

Enter name to filter on or '-' for no filtering [xml]
> -
Enter combination of types to show, R for Roots, A for Artificial Roots, N for Non-Roots [RAH]
> RAN
Enter 0x<addr> of a 'root' to show only objects owned by it or '-' for no filtering [-]
>
Enter address range in format 'M','L-U','-U' or 'L-' [0x00000000-0xFFFFFFFF]
>
Enter range of lines to print in format 'M','L-U','-U' or 'L-' [1-25]
>
Showing all objects, sorted by size.

  Addr       Size       Root-owner Parent     Total-size  Name
-------------------------------------------------------------------------------
R 0x1a490200 102,152    -          -          102,152     array of byte
N 0x1a50ecf0 65,544     0x1a355d18 0x1a89d6e8 65,544      array of byte
```

```
N 0x1a81f800 65,544     0x1a393118 0x1a393118 65,544      array of byte
N 0x1f2361b8 65,544     0x1a395418 0x1a395418 65,544      array of byte
N 0x1db84110 65,544     0x1a394918 0x1a394918 65,544      array of byte
N 0x21eccfb0 64,008     0x21de33f8 0x21de33f8 64,008      array of char
N 0x220d7c10 64,008     0x21e25100 0x21e25100 64,008      array of char
N 0x2226ef18 64,008     0x21fa2820 0x21fa2820 64,008      array of char
N 0x220ef330 64,008     0x21fffef8 0x21fffef8 64,008      array of char
N 0x222ef758 64,008     0x221e7c68 0x221e7c68 64,008      array of char
N 0x222ff168 64,008     0x22087b00 0x22087b00 64,008      array of char
N 0x223bdb60 64,008     0x222c1d10 0x21d550c8 64,008      array of char
N 0x222dfd48 64,008     0x222cf5f8 0x222cf5f8 64,008      array of char
N 0x222d0338 64,008     0x22003af8 0x22003af8 64,008      array of char
N 0x21ebd5a0 64,008     0x21dec7f8 0x21aef7c8 64,008      array of char
N 0x2200cd38 64,008     0x21eaf118 0x21eaf118 64,008      array of char
N 0x220360b0 64,008     0x221f1e60 0x221f1e60 64,008      array of char
N 0x2204ea78 64,008     0x21f356f8 0x21f356f8 64,008      array of char
N 0x22065250 64,008     0x1a355e18 0x21f1e980 64,008      array of char
N 0x2208fb78 64,008     0x21f89280 0x21d79c90 64,008      array of char
N 0x2209f588 64,008     0x1a355e18 0x21eaebb8 64,008      array of char
N 0x220aef98 64,008     0x1a355e18 0x21f55258 64,008      array of char
N 0x221b1ab0 64,008     0x1a355e18 0x21f17590 64,008      array of char
N 0x221a20a0 64,008     0x1a355e18 0x21eaf8c0 64,008      array of char
N 0x22127f50 64,008     0x1a355e18 0x21e9cda0 64,008      array of char

Displayed results  : 1-25
Matched objects    : 910,178 / 910,178
Total Size         : 127,787,448 / 127,787,448
Total Subtree Size : 3,953,833,128
```

Dumpviewer GUI and jformat

The Dumpviewer GUI and jformat are cross-platform utilities for graphically reviewing native and Java stacks as well as providing facilities to help diagnose locking and heap-related problems. On platforms other than z/OS, the tools work with platform-independent dump files extracted from the native dump. On z/OS, the tools work directly with SVC or Transaction dump, using the code from svcdump.jar.

The tools are shipped with the IBM JDKs on all platforms. The GUI is described in some detail in the PD Guides shipped with the IBM JDKs at 1.3.1 and 1.4.1.

To execute the Dumpviewer, use the following command:

```
java -Xmx512m -cp svcdump.jar com.ibm.jvm.dump.format.DvConsole -g
```

When the GUI initializes, use the File menu to locate the dump for initialization.

Restriction: For use on z/OS, you need to export the DISPLAY environment variable to a valid X Server display on a Microsoft® Windows® 32 or Linux® system.

With the Win32 JDK shipped with WebSphere on Microsoft Windows, the formatter can be invoked with the jformat command:

```
C:\Program Files\IBM\Java142\bin\jformat
```

The –g switch starts the GUI:

```
"C:\Program Files\IBM\Java142\bin\jformat" -J-Xmx512m -g
```

For more information, refer to the IBM diagnosis guides for the IBM JDKs:

- For WebSphere V5.0

  IBM® Developer Kit and Runtime Environment, Java™ 2 Technology Edition, Version 1.3.1 Diagnosis Guide, SC34-6200
- For WebSphere V5.1

  IBM® Developer Kit and Runtime Environment, Java™ 2 Technology Edition, Version 1.4.1 Diagnosis Guide, SC34-6309

You can download these manuals and detailed documentation for the Garbage Collector used in the IBM JVM from:

http://www-106.ibm.com/developerworks/java/jdk/diagnosis/

Summary

The tools described in this paper provide you with the functionality for problem diagnosis similar to the older deployment environments such as CICS and IMS, using the relatively low impact tool of unformatted dumps.

Heap-related issues such as OutOfMemoryError, other crashes, and hangs or loops within WebSphere address spaces can be analyzed using these tools.

The Svcdump.jar enables direct access to the binary SVC Dump or Transaction Dumps created on z/OS, without the need for intermediate software such as IPCS. The svcdump.jar is shipped to include:

- Dump utility

  The Dump utility (com.ibm.jvm.svcdump.Dump package) formats native and Java stacks for threads in dumped processes that include an instantiated JVM. The Dump utility includes function to print out other useful information such as in core trace buffers maintained by the JVM and the system trace, mimicing or extending the information that can be obtained with IPCS.

- FindRoots utility

  FindRoots (com.ibm.jvm.findroots.* package) provides multiple ways of formatting the object graphs present in the Java-managed heap. This is critical for the sometimes difficult tasks of finding object leaks and determining heap occupancy.

The HeapRoots utility is shipped in HR204.jar and derives from the same requirement to be able to map the heap object graphs. HeapRoots was originally developed for the JVM shipped with AIX. It is now possible to seamlessly use this code with binary SVC or Transaction Dumps, providing a range of additional function to the FindRoots utility in svcdump.jar.

The Dumpviewer GUI and jformat are cross-platform utilities for reviewing native and Java stacks. as well as providing facilities to help diagnose locking and heap-related problems. On z/OS, the tools work directly with SVC or Transaction dump, using the code from svcdump.jar.

Although we emphasize using the tools on z/OS, you are free to do the diagnosis on the platform that suits you best because the tools are Java-based.

The team that wrote this Redpaper

This Redpaper was produced by a team of specialists from around the world working for IBM®.

Rica Weller is a project associate at the International Technical Support Organization (ITSO), working in New Zealand and the USA. She worked as a Systems Engineer for S/390® for two years and as a Senior Consultant for WebSphere Business Integration on z/OS in the Competence Center with IBM Germany for three years. She also taught WBI for z/OS classes, presented at IBM and GSE conferences and co-authored several redbooks about WebSphere on z/OS at the ITSO Center. Rica holds a degree in Business Administration from TU Dresden, Germany, and a Masters Degree in Business from Massey University, New Zealand.

Clarke Dave is a Senior IT Specialist working for IBM Global Services, providing technical support for WebSphere for z/OS customers in EMEA. He has 19 years of experience in the IT industry working on z/OS and its precursors. In the past, he worked as an MVS systems programmer, a software engineer in the IBM Hursley lab service team for Java on z/OS and in a similar technical support role for the z/OS MVS BCP. This included a stint in the Poughkeepsie lab in the MVS supervisor level 2 team. He holds a Bachelor of Science in Chemistry from Durham University in the United Kingdom.

Dave Griffiths is author of the tools in svcdump.jar. He was formerly technical lead for Java on z/OS service at Hursley Lab and now works on creating RAS tools. He has 27 years experience in the IT industry mainly as a contractor for a diverse range of companies. He has been involved with Java since 1995 when he ported Java to NextStep. After a stint porting Java to RiscOS for Acorn, he joined IBM as a contractor in 1998 and finally became a regular employee in 2003.

Thanks to the following people for their contributions to this project:

Tamas Vilaghy
ITSO, Poughkeepsie Center

Notices

This document created or updated on March 6, 2005.

Send us your comments in one of the following ways:

- Use the online **Contact us** review redbook form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbook@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYJ Mail Station P099, 2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks