J Comput Virol (2005) 1: 32–43
DOI 10.1007/s11416-005-0005-6

ORIGINAL PAPER

Daniel Reynaud-Plantey

New threats of Java viruses

Received: 17 January 2005 / Accepted: 18 May 2005 / Published online: 17 September 2005

Abstract Only two Java viruses are known and have been published as a proof-of-concept: *StrangeBrew* and *BeanHive*. Since then the Java programming language has matured and greatly evolved to include a large number of new, sophisticated functionalities. At the same time, no serious study has been conducted to assess the potential viral risk attached to the new Java language. The potential threats have not really been explored yet. This article presents the state-of-the-art of Java viral capabilities and identifies some new techniques specific to the Java features that could be efficiently used by an attacker to spread malicious codes by means of Java classes. While this paper primarily focuses on targeting Java applications and not applets, the latter case will nonetheless be addressed among the different ways an actual attack could be launched by means of Java malicious codes. The protection and cure against such codes are also considered, dealing with the analysis of these programs at the bytecode level.

Keywords

1 Introduction

The mechanisms of Java viruses are an almost unexplored field. The fact is that Java is not supposed to be a virus-friendly environment and class file infection is not straightforward but remains possible. Indeed, the risk is real and cannot be ignored any longer. There are only two known Java viruses, called *StrangeBrew* and *BeanHive*. They were released in 1998 by an Australian student who wanted to show that Java virus-making was possible, so that they were created as a proof of concept; they were not meant to be dangerous and therefore never spread: But in 1998, Java was a very young language and since then a lot more developers have joined the Java community. So, does it mean that in 7-years nobody tried to create a usable Java virus? It is more than likely that someone someday used a Java virus in a real world attack, for fun or profit. Or if it did not happen yet, it sure will in the short run.

First of all, the scope of this paper is to present the capabilities of Java viruses, with no assumptions about the type of Java classes (applications, applets, servlets...), but the viruses detailed here target Java applications, NOT applets. Java applets cannot be used directly as a virus launching platform due to the security policy applied to them, but this case will be treated in this document.

This article is organized as follows. Section 2 exposes basic definitions and concepts about the Java programming language. Section 3 presents the various practical aspects of viral techniques that can be implemented specifically with Java. New techniques identified by the author are then presented. They are very likely to greatly challenge all antivirus software and must therefore seriously be taken into account. Section 4 deals with the problem of detection and cure of Java malicious codes, with a particular emphasis on low-level analysis of the techniques described in Sect. 3. Finally, Sect. 5, a few ways the actual attack could target your system or network will considere. It could take the form of a companion virus, a code dropper or a worm, to name a few.

D. Reynaud-Plantey
Ecole Supérieure et d'Application des Transmissions,
Laboratoire de virologie et de cryptologie,
B.P. 18 35998 Rennes France
E-mail: daniel.reynaud-plantey@esat.terre.defense.gouv.fr

2 Definitions and concepts

2.1 Introduction to Java

Java is a young object-oriented language designed by *Sun* (`http://java.sun.com`) with portability and security in mind. Java is portable because of its interpreted nature, the compilation step being mainly a verification and optimization process. The produced binary file can then be checked and run by the virtual machine. Therefore, Java programs can virtually run on any system for which a virtual machine is available (including Win32, most Unix systems, PDA, mobile phones, and even cars in the near future).

Basically, Java classes are defined in java files, which once compiled become binary class files. These class files are often distributed as jar (*Java ARchive*) files. End users usually associate Java with applets, which are Java programs widely used in html pages. Web browsers usually run applets in a supposedly secure environment, commonly referred to as “the sandbox”. The sandbox is a security model applied to applets which prevents them from accessing the client hard drive, network, or system. This security model has known historical problems (Mark LaDue’s Hostile applets Home Page is an interesting resource on hostile applets, though somewhat outdated) and must not be considered as the ultimate solution to applet security management.

One of the interesting features of Java is that it can be used not only to run applets but also applications (standard Java programs, which have the same privileges as traditional, native programs), servlets (server-side, CGI-like Java programs), MIDlets on cell phones and so on. Knowing this, along with the fact that Java can run on almost any platform, malicious programmers could make real cross-platform viruses.

As we will see later on, there is no direct link between the Java programming language and the binary representation of class files: “The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format” (Tim Lindholm and Frank Yellin). Though this point of view is not commonly shared, it is possible to consider Java as just a convenient front-end for developers and the compiler as a translator between Java and the class file format. The real language is contained in binary class files, and is more powerful than Java. Virus examples in the following sections use some of the tricky aspects of the class file format to demonstrate that malicious programs could bypass the built-in security of the virtual machine.

An important aspect of Java security relies in the virtual machine and the way it loads and verifies classes. Indeed, binary Java classes must comply with many static (is the class file ok?) and structural (are the instructions valid?) constraints. They must also undergo a data-flow analysis (is the interaction between instructions valid?) before they can be executed. This verification process is a really strong constraint for virus-makers and it certainly makes the virus-making process a lot more complex, because errors when tampering with class files irremediably lead to verification failures, that is to say no execution of the broken class.

2.2 The class file format

Class files can be infected by other class files, similar to what happens with PE or ELF infection. The class file format is defined and fully documented in Sun’s book, *The Java virtual machine specification* by Tim Lindholm and Frank Yellin . Here is an excerpt from the above specified format that reviews the class file format (where ux represents x unsigned bytes):

```
ClassFile {
u4 magic;
u2 minor_version;
u2 major_version;
u2 constant_pool_count;
cp_info constant_pool[constant_pool_count-1];
u2 access_flags;
u2 this_class;
u2 super_class;
u2 interfaces_count;
u2 interfaces[interfaces_count];
u2 fields_count;
field_info fields[fields_count];
u2 methods_count;
method_info methods[methods_count];
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

Here is what the different items mean:

- magic: four bytes having the hex value 0xCAFEBABE.
- minor_version and major_version give the version number of the class file format used in this file.
- constant_pool_count, interfaces_count, fields_count, methods_count and attributes_count give the number of elements in each table.
- access_flags is a mask of flags giving the access properties of this class (public, private, abstract. . .)
- this_class and super_class are two indices in the constant pool pointing to information regarding this class and its super class.

The most important items are the five following tables:

- **constant_pool**: this table lists all constants used in this class (such as methods and class names, strings, initial variable values. . .).
- **interfaces**: at list a the interfaces implemented by this class.
- **fields** and **methods**: information regarding the fields and methods defined in this class. The code of each method and constructor is contained in the methods table.
- **attributes**: as expected, the attributes of this class file (for example, the SourceFile or Deprecated attributes).

In our case, the virus will more specifically be interested in the constant_pool, fields and methods tables. The methods table is made of method_info structures, as appearing in Tim Lindholm and Frank Yellin :

```
method_info {
u2 access_flags;
u2 name_index;
u2 descriptor_index;
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

The fields in a method_info structure are self-explaining enough, there is no need to comment on them. Most of the useful information is contained in the attributes table of the method, particularly the Code_attribute:

```
Code_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 max_stack;
u2 max_locals;
u4 code_length;
u1 code[code_length];
u2 exception_table_length;
{ u2 start_pc;
u2 end_pc;
u2 handler_pc;
u2 catch_type;
} exception_table[exception_table_length];
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

The Code_attribute contains a code array that actually lists the Java virtual machine instructions for this method. This is the array the virus will have to locate and modify, for each targeted method.

3 Practical aspects of Java viruses

This section presents some general-purpose class file infectors written in Java. As they represent potential threats if they fall into wrong hands, their source code has not been made public. However, computer security professionals can obtain them upon request to the author. The source codes have been provided to the EICAR group (`http://www.ei\-car.org/`) for release to the different antivirus software publishers. This part is organized as follows : Section 3.1 presents a new virus which is built on the same concept as *StrangeBrew* but tries to be more efficient and more powerful. Section 3.2 then presents a new virus concept that tries to adapt the idea of code encryption to the Java viral world. Finally, Sect. 3.3 discusses miscellaneous stealth techniques that could be employed by Java viruses, including a quite tricky one that allows the use of invalid identifiers in class files.

3.1 A basic appending virus

The goal of our virus is to append its code to a target file, where "code" means a set of constants, fields and methods. Basically, our virus has to append its own constants, fields and methods to the constant_pool, fields and methods tables of the target, and then change constant_pool_count, fields_count and methods_count accordingly. In order to become executable, the virus then has to insert a call to viral methods in the main method of the target class.

Let's take a real example. To illustrate how to make a general-purpose Java virus, we are going to introduce a new virus called *JavaVxConcept*. It originally compiled to *JavaVxConcept.class*, let's assume it infected a file called *LocalClass.class*. If *LocalClass* is run, *JavaVxConcept* goes through the following steps:

1. **Find a target**. For security purposes, *JavaVxConcept* looks for a file named LocalClass.class.cfg containing the name of a valid target, for example Victim.class.
2. **Retrieve information about the target**. Such as its method_count and constant_pool_count.
3. **Retrieve information about the local (viral) file**. Finding itself is not straightforward, but many solutions can be implemented. *StrangeBrew* searches the working directory for an infected file to retrieve the viral information from *JavaVxConcept* knows the name of the infected class it is running from because of its local_filename field, set by its parent virus.
4. **Mark the target as infected**. *JavaVxConcept* does so by adding 0×1000 to its access flags. 0×1000 is not a valid access flag, but Tim Lindholm and Frank Yellin () specifies that "All bits of the access_flags item not assigned, should be ignored by Java virtual machine implementations".
5. **Insert the viral methods in the target**. This step occurs in two phases:

   (a) First, it increments the target's methods_count and inserts the viral methods at the end of the methods table of the target. To find the viral methods, *JavaVx-Concept* looks for the n last entries of its own methods table, where n is the number of viral methods, stored in the nb_viral_mt field. This field is set by the original virus during the first infection.
   Actually, the viral methods have to be modified before they can be appended to the target's methods table. The instructions in the viral methods contain indexes that refer to the local constant pool, so that these indexes have to be changed in order to be valid in the target file. This step is quite complex, let's see why with a small example: suppose the viral method viral() contains the instruction ldc 0×42. It tells the virtual machine to load the constant at index $0{\times}42$ in the constant_pool, for example the string "All your base". The problem is: once inserted in Victim.class, "All your base" will probably not appear at index 0×42. If it appears at index $0 \times A2$ everything is perfect, the virus just has to replace ldc 0×42 with ldc $0 \times A2$. But suppose the string appears at index $0 \times 29A$ (or any number greater than $0 \times FF$). The virus cannot just replace ldc 0×42 with ldc $0 \times 29A$ because ldc takes a single byte as an argument and $0 \times 29A$ needs 2 bytes to be written. In this case we have to replace ldc 0×42 with ldc_w 0×02 $0 \times 9A$ (ldc_w takes a 2-bytes argument).
   A new problem arises here. Suppose viral() contains the following instructions:
   ```
   ifeq 0x00 0x05
   ldc 0x42
   pop
   ```
   This sequence means "if the top value on the operand stack equals 0, pop it (the jump after the ifeq instruction points five bytes after its address). Else,

load constant at index 0×42 and then pop the top value on the operand stack".
As we have seen above, we might have to replace ldc 0x42 with ldc_w 0×02 $0 \times 9A$. The sequence of instructions becomes:
ifeq 0x00 0x05
ldc_w 0x02 0x9A
pop
This sequence will result in an error because the jump after the ifeq instruction points to the $0 \times 9A$ byte, which is not the valid address of an instruction. This is normal because we expanded the data between the jump instruction and its target. The virus thus has to change the jump value accordingly. The correct sequence should be:
ifeq 0x00 0x06
ldc_w 0x02 0x9A
pop
This problem is quite complex, because the offset after an if condition, a tableswitch or a lookupswitch is signed, and the arguments of tableswitch and lookupswitch must be four byte-aligned.

(b) Then it modifies the main method of the target so that it can execute the virus. This is done by inserting three bytes in the code array of the Code_attribute attribute in the target's main method. The three bytes are invokestatic, indexbyte1 and indexbyte2. The last two bytes are considered as an unsigned short point to the infectTarget() method in the target's constant_pool. The code_length in the Code_attribute is then incremented.

6. **Insert the viral fields in the fields table** and increment the target's fields_count item. *JavaVxConcept* retrieves the viral fields the same way as it retrieved the viral methods: the field nb_viral_ft tells it to retrieve the last *n* entries of its local fields table.
7. **Append the viralconstant_pool items** to the end of the target's constant_pool and increment its constant_pool_count item. Once again, the viral constant_pool items are the last nb_viral_cp entries of the viral file.

Once all these steps have been successfully completed, the resulting class file is a valid class containing the original victim code plus the *JavaVxConcept* viral code. If you run the infected application, *JavaVxConcept* will first proceed from step 1 to 7. Once it is done, the original program runs normally.

JavaVxConcept is a basic virus, rather slow and inefficient because an antivirus can easily detect and stop it. Once a class file is infected, it is almost impossible to remove the code attached to it, because *JavaVxConcept* inserts itself in three different places and modifies a lot of data. As such, *JavaVxConcept* is not very likely to be used in an attack, but it is useful to explore the way class files can be infected and it can be used as a basis for more efficient viruses. Indeed, one of its main assets is that developers do not need to care about the underlying viral mechanisms, *JavaVxConcept* is entirely dynamic. For example, any method or field can be added and the virus will function normally once recompiled, which seems to be normal but not that obvious. As a reminder, *StrangeBrew* allowed only one viral method, no field, and if the method were changed, some hardcoded fields (such as the length of the viral data) have to be changed manually. Therefore, this virus is a convenient platform for the study of more complex viral algorithms.

3.2 Code encryption

We have seen that the previous virus has many limitations for which we could try to find a workaround. The main problem is that it copies a lot of data into its target, and these data are almost always the same from one copy to the others (only indexes change). Moreover, the virus parses its own code before inserting it and therefore if an error occurs in a given virus, it will also appear in all its children. The probability that an error will occur increases in important as the virus spreads, so it will probably stop working after a certain number of infections, depending on the quality of the virus. Add this to the fact that the Java bytecode verifier accepts no error, even the subtler ones, and that the virus can potentially encounter different virtual machines with different checking procedures, and you will understand that making an absolutely error-free Java virus is a tough task.

We now consider a more sophisticated proof-of-concept virus, denoted *JavaVxCryptoConcept*. A possible solution to the above-mentioned problems and limitations is to include the virus as a whole in the target. Once infected, the target loads a new class with the viral data it contains, which is identical to the original virus. The infection process can then repeat itself, with a smaller likelihood to fail.

However, problems remain with this approach. First of all, where should the virus be included in the target? We need to insert a lot of data that should not get checked by the bytecode verifier but can still be easily accessible by the target, once infected. The simplest way to do this is to store the whole virus as a Constant_Utf8 (string) structure in the target's constant_pool. In order to be harder to analyze and to detect, the virus could store its own code as an encrypted string with a different key each time. I refer to this technique as code encryption, to highlight the parallel between it and the existing packing techniques for executable files.

A last point needs to be mentioned: the virus has to load a new class, based upon the encrypted data it contains. In order to define this class on the fly, two options are possible. The simplest is to write the decrypted data to a class file and then instantiate the virus normally:

```
RandomAccessFile out = new
RandomAccessFile
("JavaVxCryptoConcept.class", "rw");
out.write
(decrypted, 0, decrypted.length);
out.close();
JavaVxCryptoConcept
v = new JavaVxCryptoConcept();
```

This approach might not work in some cases, depending on the classpath configuration and on the virtual machine: if the virtual machine does not look for classes in the working directory or if the working directory is not in the classpath, then the newly created class *JavaVxCryptoConcept* will not be found. The solution could be to look for a directory in the classpath where *JavaVxCryptoConcept.class* can be written. Yet, writing the data to a file is not a really good idea because the decrypted data can then be easily analyzed by an antivirus, and the advantage of encrypting the data beforehand is then lost. Using the file system adds an unnecessary step: according to the Java 1.5 API Specification, "[. . .] some classes may not originate from a file; they may originate from other sources such as the network, or they could be constructed by an application. The method ClassLoader.defineClass converts an array of bytes into an instance of class Class [. . .]". The following piece of code does the same as above, but without using the file system:

```
Class decoded = defineClass("JavaVxCryptoConcept", decrypted, 0, decrypted.length);
Constructor con = decoded.getConstructor();
Object obj = con.newInstance();
```

Unfortunately, ClassLoader.defineClass is protected, so we need to subclass ClassLoader in order to be able to use it. To sum it up, the new virus goes like this:

1. if class JavaVxCryptoConcept is run directly, **read the local data** (the whole JavaVxCryptoConcept.class file) and create a new JavaVxCryptoConcept instance, passing the data as an argument. Else, if class JavaVxCryptoConcept is instantiated, **look for a target** in a file called `target.cfg`;
2. **retrieve information about the target**. If the target's superclass is not Object, stop infection here;
3. **add the constants** necessary for the execution of the decode() method to the target's constant_pool;
4. **encrypt the viral data**, convert the encrypted data to ascii and append it to the target's constant_pool as a Constant_Utf8 entry. In order to obtain a different encrypted string each time, the algorithm and the key can change. JavaVxCryptoConcept uses the target file name as a key and the following method to encrypt itself:

```
// import javax.crypto.*;
// import javax.crypto.spec.*;

byte[ ] code(String key)
throws Exception {
// add new algorithms here
String alg[ ] = {"DES", "DESede"};

int num = key.length(){\%}2;

KeySpec s = (num==0 ? new DESKeySpec
            (key.getBytes()):
            new DESedeKeySpec
            (key.getBytes()));

SecretKey k = SecretKeyFactory.
            getInstance(alg[num]).
            generateSecret(s);}
Cipher c = Cipher.getInstance
(alg[num]);
c.init(Cipher.ENCRYPT_MODE, k);
return c.doFinal(data);
}
```

5. **change the value of target's super_class** so that it points to ClassLoader. This step has to be completed if we want the target to be able to define the *JavaVxCryptoConcept* class on the fly. It is also used as a marker for the infection: once ClassLoader is set as the target's superclass, it will never be infected again (see step 2);
6. **insert the decode() method** in the target's method table;
7. **insert a call to decode()** in the target's main method. The decode() method basically does the following:
   (a) **read the local constant_pool** and look for the last Constant_Utf8 entry;
   (b) **decrypt this Constant_Utf8 entry**;
   (c) **load the new class** defined by the decrypted data, as seen above and instantiate it. Infection then proceeds again at step 1.

JavaVxCryptoConcept is the combination of two techniques: minimal code appending, as seen in the previous example, and code encryption. Most constant parts and errors are likely to come from the code appending part. Code encryption makes the virus harder to detect and analyze. The decryption method can be easily reverse engineered, but code encryption coupled with some anti-reverse techniques could disturb the virus analysis. The study of these techniques is beyond the scope of this article because they are not specific to Java viruses, but some stealth techniques are presented in the following section.

3.3 Stealth techniques

Hiding information in class files We have seen with the previous section that code could be packed and inserted in the constant pool as a Constant_Utf8. There are many other places in a class file where information can be stored without being checked. This can be achieved with an attribute structure, which has the following format:

```
attribute_info {
u2 attribute_name_index;
u4 attribute_length;
u1 info[attribute_length];
}
```

As you can see, an attribute is made to store variable-length data. Moreover, attributes can be found at many locations: in the ClassFile structure itself (remember the table attributes), in a field_info, a method_info or a Code_attribute structure. And the most interesting feature of attributes: "A Java virtual machine implementation is required to silently ignore any or all attributes in the attributes table of a ClassFile structure that it does not recognize" (Tim Lindholm and

Frank Yellin). Sun has made this choice with extensibility in mind, to ensure backward compatibility with older VMs in case new attributes are defined. To sum it up, attributes are interesting for storing information such as packed code for the following reasons:

- They allow the storage of arbitrary, variable-length data.
- The data will not get checked.
- The virtual machine is required to silently ignore unknown attributes.
- The stored information will not appear in decompiled or disassembled class files.

The turnover is that access to that information needs more work than accessing information stored in the constant pool. For example, imagine the following constants are appended to the constant pool of the target:
cp[x] -> CONSTANT_String: string_index = x+1
cp[x+1] -> CONSTANT_Utf8: value = "...packed code..."

A method can then easily access it with the instruction ldc x, which will push the value "...packed code..." on the operand stack. On the other hand, to retrieve the information stored in an attribute, the program must do the following: open the local class file (as seen above, this can be an issue) and parse the whole class file until the attribute is found.

There is another interesting way to hide information in a class file, using a trickier approach. The idea is to store the data directly in the code array of the Code_attribute of a given method, after the last return or ret instruction. This is tricky because the code arrays are highly sensitive parts of the class file and extensive checks are being performed on it. Storing information (not necessarily instructions) there remains however possible because of the dead nature of the inserted information. They are not going to be executed, so they just have to comply with some structural constraints (for example they must be valid instructions with valid arguments) and they will never undergo the data-flow analysis, which could determine that these instructions are illegal though well-formed. The interest of storing information there is limited but can be used to confuse reverse engineers.

Consider the following simple class:

```
class VeryDummy {
public static void main(String[ ] arg) {
System.out.println
("doing absolutely nothing...");
}
}
```

The command `javap -c VeryDummy` gives the following output for method main:

```
public static void main
(java.lang.String[ ]);
Code:

0: getstatic #2;
//Field java/lang/System.out:
Ljava/io/PrintStream;

3: ldc #3;
//String doing absolutely nothing...

5: invokevirtual #4;
//Method java/io/PrintStream.println:
(Ljava/lang/String;)V
8: return
```

Now suppose we add the information 61 6C 6C 79 6F 75 72 62 61 73 65 (the hex values for "allyourbase") after the return instruction in VeryDummy.class and change code_length and attribute_length accordingly. The class can still be executed with no apparent change, which means that the code array has been checked and declared valid. Let us disassemble VeryDummy again, javap -c VeryDummy now gives the following output:

```
public static void main
(java.lang.String[ ]);
Code:

0: getstatic #2;
//Field java/lang/System.out:
Ljava/io/PrintStream;
3: ldc #3;
//String doing absolutely nothing...

5: invokevirtual #4;
//Method java/io/PrintStream.println:
(Ljava/lang/String;)V

8: return
9: ladd
10: idiv
11: idiv
12: lshl
13: ddiv
14: lneg
15: frem
16: fadd
17: ladd
18: drem
19: lsub
```

The information we added appear as valid instructions which mean nothing. The interest is that the modification is invisible in decompilers, which perform data-flow analysis. The output of jad -nonlb -noctor -p VeryDummy.class is still:

```
import java.io.PrintStream;
class Test {
public static void main(String args[ ]) {
System.out.println("doing absolutely nothing...");
}
}
```

However, jad provides the -dead option to decompile dead parts of code. In our case, jad -dead VeryDummy.class causes jad to crash. This is not surprising as the instructions jad tries to decompile are illegal ones. In conclusion, adding information to code arrays is not necessarily efficient and can be error-prone if not used correctly but can also be confusing for reverse engineers.

Method name stealing As stated in the introduction of this document, there is no direct link between the Java programming language and the class file format. This fact becomes obvious when you work with bytecodes. Some aspects of the difference between Java and the bytecode can be useful to virus makers, an example of such difference is going to be detailed in this section.

Invoking a given method SomeClass.someMethod() from Java requires the knowledge of three characteristics:

– the class defining the method, SomeClass.
– the name of the method, someMethod.
– the arguments and return type of the method, called the descriptor of the method.

The bytecode knows (almost) nothing of these characteristics. From the bytecode point of view, the invocation of a given method requires a call to the invokevirtual, invokespecial or invokestatic instructions immediately followed by the *index* of the method in the constant pool. The role of the bytecode verifier is to check that the method has a valid name, is accessible, is called with the appropriate arguments, etc... But the fact is that from the bytecode point of view, a method is an index in a constant pool. So you can wonder whether it is really useful for a virus which operates directly at the bytecode level to add a *whole* method in the target. By whole method I mean the method_info structure containing the code of the method, but also the constant pool entries associated with it, including its name, descriptor and necessary runtime constants (such as pointers to other methods). Actually, adding the name of the method along with the method itself is of no use, since the viral methods are not supposed to be reused or accessed from outside the infected class. Including the real name of a method (such as "decode" in *JavaVxCryptoConcept*) will only give away unnecessary information to reverse engineers.

To prevent this information leak, a virus could use method name stealing. This technique consists in reusing the name of a method already defined in the target. This is valid as long as the descriptors of the original method and the viral one are different. For example, consider an improvement of *JavaVxCryptoConcept*:

– The decode(void) method could now be defined as decode(java.lang.Void).
  This is done to ensure that decode and the target method will have different descriptors, because java.lang.Void is never used in real java programs. If it is used in the target, the virus maker should consider that he definitely has a bad karma. A call to decode should now look like decode((Void)null).
– When parsing the target file and more particularly its constant pool, the virus should keep in memory the index of the name of a method declared in it. In case no other method is declared, main can always be used.
– When adding the viral constant pool entries (necessary for the execution of the viral method), the symbolic reference to the viral method must be updated. This symbolic reference is a CONSTANT_Methodref_info:
```
CONSTANT_Methodref_info {
u1 tag;
u2 class_index;
u2 name_and_type_index;
}
```
  name_and_type_index is also the index of a structure is the constant pool:
```
CONSTANT_NameAndType_info {
u1 tag;
u2 name_index;
u2 descriptor_index;
}
```
  name_index is the index of a Constant_Utf8 structure giving the name of the method. By changing name_index to another value so that it points to another Constant_Utf8 string, the method will almost be allowed to be called with its new name. Almost, because the name_index in the method_info structure of the viral method must also point to the new name.

An example will show the expected result. Let A and B be two victim classes:
```
class A extends java.lang.Object{
A();
public static void main(java.lang.String[ ]);
}
class B extends java.lang.Object{
B();
public static void main(java.lang.String[ ]);
private void doSomething();
}
```
Now suppose a version of *JavaVxCryptoConcept* implementing the method name stealing technique is run and targets these two classes. The output of javap -private A B could be:
```
class A extends java.lang.Object{
A();
public static void main(java.lang.String[ ]);
private void main(java.lang.Void); // former decode()
method
}
class B extends java.lang.Object{
B();
public static void main(java.lang.String[ ]);
private void doSomething();
private void doSomething(java.lang.Void); // former
decode() method
}
```
This simple technique prevents adding unnecessary information into the victim's constant pool, and can be confusing

if it has not been clearly identified for the virus analysis. However, if the goal of the virus maker is to make the reverse engineering of his malicious program harder, he could use more powerful anti-reverse engineering techniques, which are beyond the scope of this article. Such techniques have been presented at the Virus Bulletin 2005 Conference in Dublin (Daniel Reynaud-Plantey (2005)) and will be the subject of a future research work.

Invalid identifier injection This technique is derived from the method name stealing technique. It is based on the fact that some virtual machines do not check if the names of classes, fields and methods are valid Java identifiers (including Sun's Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08 and 1.4.1_02-b06) and Kaffe). The check is only made at compile-time, contrary to what is stated in Tim Lindholm and Frank Yellin : "The checks performed [. . .] include [. . .] checking that all field references and method references in the constant pool have valid names". This has already been pointed out in Godfrey Nolan and Crema, one of the first commercial obfuscators, uses it to perform name mangling. However, this technique seems to have been overlooked since it violates the specification and it still has not been fixed.

Let's consider a simple example illustrating what can be done with this technique: we are going to forge a class with an empty name.

We are first going to create the class with a standard name (Test):

```
class Test {
String s = "";
public static void main(String arg[ ]) {
System.out.println("how are you gentlemen?");
}
}
```

The empty string is here to ensure that the constant pool will contain an empty Constant_Utf8. Using a disassembler like ByteCodeTester, we obtain the following output:

```
magic number = CA FE BA BE
bytecode version = 49.0
constant pool count = 35

cp[1] (offset 0xa) ->CONSTANT_Methodref:
class_index = 8;
name_and_type_index = 19

(...)

cp[9] (offset 0x1f)->CONSTANT_Methodref:
class_index = 25;
name_and_type_index = 26

cp[10] (offset 0x24) -> CONSTANT_Class:
name_index = 27
cp[11] (offset 0x27) -> CONSTANT_Class:
name_index = 28

(...)

cp[19] (offset 0xa4) ->
                    CONSTANT_NameAndType:
                    name_index = 11;
descriptor_index = 12

cp[20] (offset 0xa9) -> CONSTANT_Utf8:
                           value = ""

cp[21] (offset 0xac) ->
                    CONSTANT_NameAndType:
                    name_index = 9;
descriptor_index = 10

(...)

cp[26] (offset 0xd6) ->
                    CONSTANT_NameAndType:
                    name_index = 33;
descriptor_index = 34
cp[27] (offset 0xdb) -> CONSTANT_Utf8:
                           value = "Test"
cp[28] (offset 0xe2) ->
                    CONSTANT_Utf8: value =
                    "java/lang/Object"

(...)

access flags = 0x20 [ ACC_SUPER ]

this_class index = 7
super_class index = 8

(...)
```

We can see that this_class index is 7. The constant pool entry number 7 is a Constant_Class structure which contains the value name_index = 27 ("Test" in this case). Just replace this name_index with 20 and the Test class will have an empty name. In order to be able to execute it, "Test.class" has to be renamed ".class". We can now execute it with the command java "". javap "" gives the following output:

```
Compiled from "Test.java"
class extends java.lang.Object{
// empty name for the class !
java.lang.String s;
(); // constructor of the class

public static void main
(java.lang.String[ ]);
}
```

Before the modification, javap Test gave:

```
Compiled from "Test.java"

class Test extends java.lang.Object{
java.lang.String s;
Test();
public static void main
(java.lang.String[ ]);
}
```

A malicious program could use this technique in order to hide classes on UNIX systems. It gives the attacker the opportunity to use tricky names for his classes, allowing him to hide them more easily and maybe to exploit bugs in class loaders. In addition to this, he could use invalid identifiers for his viral attributes and methods, making decompilation useless for the analysis of his malicious program without modifications.

4 Detection and cure of Java viruses

4.1 Detection with signatures

The generation of a signature for a Java virus should be as hard (or easy) as the generation of a signature for a native virus. A basic virus like *JavaVxConcept* presented in Sect. 3.1, contains large parts of constant data which can be easily used as a signature. However, *JavaVxCryptoConcept* significantly reduces the amount of constant parts and should therefore be harder to identify with a signature. The static analysis has to detect the virus based on the analysis of the following data:

1. The entries that have been added to the constant pool (mainly for the execution of decode()). In this case, they are about 160 but do not characterize the virus. Moreover, it is possible to optimize *JavaVxCryptoConcept* so that there are less entries. This can be achieved by removing unnecessary operations from decode(), by inlining methods and by avoiding adding redundant entries (for example, it is possible to avoid adding a CONSTANT_Utf8 entry pointing to java/lang/Object because the target is most likely to contain it). It is also possible to randomize the order in which these constants are added.
2. The encrypted viral data, it is stored as a fixed-length variable hex string, but once again it is possible to add some entropy by using variable compression before the encryption of the data and by using different encodings (hex, base64 or other MIME encodings) for the encrypted data.
3. The decode() method itself, it is a 552 bytes long array in which indexes change according to the target but which mostly remains constant. It is however possible to use a simple obfuscation algorithm to modify the bytecodes before inserting decode() in the target.

On the whole, *JavaVxCryptoConcept* shows that it is possible to create a Java virus with fewer constant parts and could become smaller and almost totally self-modifying with a few optimizations. More in-depth experiments should be run but it is likely that generating a signature for middle-level viruses such as *JavaVxCryptoConcept* will be tricky.

4.2 Detection with code emulation/dynamic analysis

This aspect is currently under study. It is not clear whether present behavior analysis techniques, code emulation techniques and more generally dynamic analysis would be efficient at detecting such Java malicious codes as we have presented above. Experiments performed at the Virology and Cryptology Laboratory proved that none of the available antivirus software — irrespective of their set-up, — was able to detect them.

4.3 Analysis of the malicious code

This problem is not as easy as it may seem at first sight. A thorough study of analysis techniques is beyond the scope of this article but a detailed insight can be found in Daniel Reynaud-Plantey (2005). Here is a brief review of the problems the analyst might encounter :

1. The malicious code can be included in a **"protected" jar file**. Actually, the file containing the malware does not even necessarily have the .jar extension. Consider the following example:
   ```
   >java Test
   I'm just a poor test...
   >echo Main-Class: Test> manifest.mf
   >jar cmf manifest.mf test.jar Test.class
   >java -jar test.jar
   I'm just a poor test...
   >mv test.jar test.jpg
   >java -jar test.jpg
   I'm just a poor test...
   ```
   This simple example shows that Java classes can be executed from any jar file, no matter what their name is. It is also possible to forge a jar file from which the classes cannot be extracted directly, thanks to the invalid identifier flaw explained in Sect. 3.3. This level of protection is meant to waste the time of the analyst before he can even start reverse engineering the malware.
2. Once the classes are extracted, the analyst will probably try to **decompile** them. The advantage of Java for reverse engineers is that by default it is easy to recover the source code of a class given only its compiled form. However, it is pretty easy to prevent decompilation. There are many levels of protection, from layout(modification of identifiers, possibly invalid) to control flow transformations and standard obfuscation. These transformations are detailed in Christian Collberg's paper entitled *Taxonomy of Obfuscating Transformations* (Godfrey Nolan). Specific programs have even been created to defeat decompilers, such as HoseMocha, which can be found on Mark LaDue's Hostile Applets Home Page.
3. Using a **debugger** on obfuscated class files is also quite hard. It is possible to make the use of a debugger even harder, for example by using multithreading. However, some debuggers can be particularly adapted to viral files, such as omniscient debuggers.

4. The last solution is the low-level analysis of the malware through **disassembling**. Layout transformations do not have a huge impact on the disassembling process, but control flow transformations do and a good obfuscator can produce really tricky bytecodes. This is, however, the only solution left to the analyst in case the virus uses armouring techniques.
5. In the case of an encrypted virus like *JavaVxCryptoConcept*, the analyst will first have to reverse engineer the decode() method in order to see how the decryption occurs before being able to proceed to the analysis of the virus itself. *JavaVxCryptoConcept* only uses encryption to prevent automatic detection and therefore will not resist a good analysis, because the encryption key is contained in the file (it actually is the file name). But Eric Filiol (2005) shows that by using environmental key generation, the analysis of the virus is at least as hard as the cryptanalysis of the underlying encryption scheme.

5 Real world attacks

Some techniques discussed below are examples of the way a Java viral attack could infect you and settle on your system. In the last parts, the described techniques could be used by Java viruses to push the attack further, and actually use a Java virus as a platform for a broader attack. To launch a real attack, the malicious programmer has two advantages. Java is everywhere and it is possible to find systems equipped with a virtual machine that the user or the administrator did not notice. Also, most people regard Java as a secure platform and have a tendancy to trust Java code. Such trust relationships are dangerous and may lead to attacks that should otherwise be easily avoided.

5.1 Exploits and worms

For the moment, most people think of Java security in terms of applets and mobile code execution. It is true that for the average user, Java is only used for small applications such as applets or MIDlets which can only run in a "controlled" environment. The danger is to think that Java cannot harm your system because of this sandbox model. When *StrangeBrew* was released it was common to hear things such as "don't worry, applets run in a sandbox and therefore can't do anything dangerous". Many people are still probably thinking like this. This kind of assertion is FALSE. Hackers have proved capable of finding flaws in virtual machines and security model implementation, and most importantly they showed they could exploit them as we shall see later on. The fact is that those high risk vulnerabilities have not been exploited very smartly and most of them got fixed pretty quickly. However, it is possible that some people, did detect flaws but chose not to disclose them, exploiting them for their own use.

So How then to explain that even with very important stakes and millions of vulnerable systems there have been no large scale attacks? A possible explanation is that the hacker community is not used to Java (note that this is the same for the security professionals community) and most potential Java hackers probably prefer exploiting some buffer overflow than trying to bypass the strong constraints of the Java security model because of the expected amount of work.

A very interesting trojan known as *Java.Classloader* (alias *Java.Byteverify* or *Trojan.ByteVerify*) exploits a flaw in all Microsoft's VM versions earlier than 5.00.3810 (Microsoft Security Bulletin MS03-011). The problem resulted from the way the bytecode verifier checked the code array when a class was loaded. This vulnerability enabled the attacker to run a "fully trusted" applet, with the same permissions as the user who accessed the page containing the applet. A version of this exploit seems to have been used by Russian hackers to target Australian bank site users. The applet was loaded with the following tag:

```
<applet archive="classload.jar"
code="GetAccess.class"
width="1" height="1" param
name="ModulePath"
value="http://aicworld.info/msxmidi.dat">
```

Once loaded, the applet downloads the msxmidi.dat file, renames it to msxmidi.exe and executes it. More details about this trojan can be found in http://jola.clover.com.au/pipermail/polonet-l/2004-April/000049.html and http://www.auscert.org.au/3981.

Another vulnerability called BOHTTPD (CERT) targeted Netscape Communicator. It is now outdated but illustrates the way an applet can jump out of the sandbox on a specific platform. Under normal conditions, an applet can only initiate connections to the server it was loaded from and cannot listen on a port. BOHTTPD enabled the attacker to violate these rules. It could have many uses:

1. map a network or portscan machines behind a firewall in a corporate network,
2. launch a DDOS,
3. bounce connections to relay network attacks.

These exploits, coupled with the viral techniques demonstrated above, could wreak havoc if a viral applet was implanted on newly defaced websites. But despite this danger, they cannot be used directly to make Java worms. Let's examine some ways a Java worm could spread:

1. Web **applets** are not good worm vectors because of their client oriented nature. Its means that the attacker is passive and can only wait for a victim to visit the site containing the malicious applet. And even in that case, the infection process is not the one of a classic worm, because the infection will not spread from one infected machine to another.
2. E-mail **applets** look more interesting in order to make a worm. However, most e-mail clients will not allow Java applets to run in html mail by default, so the malicious Java applet could only run if another flaw were exploited.

3. **Servlets** can probably be exploited. But the flaw will most probably not be specific to Java, it is more likely that the flaw will come from the servlet running engine (such as Tomcat for Apache (The Apache Jakarta Project)). It is however theoretically possible to find flaws similar to the ones encountered with PHP or CGI scripts. Once again, the small number of exploits targeting servlets is probably linked with the fact that few people take the time to find them.
4. **MIDlets** are roughly the equivalent of applets for cell phones. Though the idea of a Java worm for cell phones is appealing, it seems that such a malicious program would encounter the same difficulties as web applets. However, more research needs to be conducted by the security community before any assertions about the strength of cell phone security can be made. Adam Gowdiak has already demonstrated that powerful attacks could be implemented against Nokia phones. (Java 2 Micro Edition (J2ME) security Vulnerabilities)

In conclusion, Java is at the same time interesting for exploit-writers (because of its wide multi-platform use) and time-consuming because of the built-in security of virtual machines. Even if this security model is not perfect, it shows that Java has been created with security in mind and this is probably one of the reason why there have been no Java worms for the moment.

5.2 Companion code

Java can be used to create malicious programs mimicking native companion viruses.Creating a simple Java companion virus targeting standalone applications (that is to say class files containing a main method) is trivial. A more interesting and stealthier approach could target jar files. Consider the following algorithm:

1. Search for a jar file in the current directory.
2. When a jar file is found, look for the main class in it. The main class of a jar file can be specified in the Main-Class section of the manifest.mf file in the jar file. If this section does not exist, the main class can be found by looking for class files with a main method in the jar file. Note that not all jar files contain main classes.
3. If a main class has been found, copy the virus (the current class) in the jar file with the same name as the target class. The jar file being basically a zip file, it is possible to have duplicate entries in it. The main class of the target jar file is now the virus.
4. Finally, attempt to load the original class that the virus has hijacked, that is to say the second class named like the current class in the current jar file, so that the user sees no difference with a normal execution.

These are the basic steps that a companion virus could implement in order to infect jar files.

Another approach can take advantage of the object-oriented nature of Java. Classic companion viruses use the tricky aspects of execution priority, playing with file extensions and the PATH environment variable. Java classes necessarily have the .class extension so this approach is not possible, the interesting aspect is class loading priority. Virtual machines generally use the CLASSPATH environment variable, which is the same as PATH but for Java classes.

In the last example, the virus was used to run malicious applications before the legitimate ones, do the viral work and then execute the legitimate applications (here, classes are considered as executables). With the new concept; the malicious code would not run before the real one, it would replace some classes with malicious ones (thinking of classes in terms of objects). This approach is considerably powerful because almost everything is accessible from Java as a Java class. For example, even a string is an object from the Java point of view (cf java.lang.String). These classes, in the Java API are called the bootstrap classes. It can be particularly powerful to load malicious classes instead of the real bootstrap classes.

Most virtual machines allow for the redefining of the bootstrap classpath. For example, consider the following batch script for Windows:

```
: java.bat
@echo off
java.exe -Xbootclasspath/p c:\malicious.jar %*
```

If this script is saved in the PATH before the java.exe executable, it will silently tell the virtual machine to first look for bootstrap classes in c:\malicious.jar (this can be done with a simple alias under UNIX systems). Here are some examples of interesting classes to override:

1. java.lang.ClassLoader: could give more control over the class loading process
2. java.net.Socket: could allow for man in the middle attacks. For example, the fake Socket class could be told to connect to a proxy instead of the requested IP.
3. java.util.Random and java.security.SecureRandom: could be used to generate cryptographically weak random number sequences.
4. javax.crypto.KeyGenerator and SecretKeyFactory: could be used to log the generated keys

The only limit to this technique is the imagination of the attacker. It is particularly powerful because there is no need to tamper with class files and the modification is invisible to the targeted application.

5.3 Code dropping

That the sandbox model should not be trusted has been demonstrated. However, even when the security model is bypassed, the Java code can hardly make low level attacks because it has only a restricted access to the system. The idea is that Java can be used as a vector for more powerful attacks, where the process is the following. First, run Java untrusted mobile code (such as an applet, a servlet or a midlet). Then, jump out of the sandbox by exploiting a flaw and finally drop native code.

These steps looks like the way the Java.ClassLoader trojan acts, but it is not exactly code dropping. The idea of code dropping is to store the native code directly in the class file, for example as an encrypted string constant (just like *JavaVxCryptoConcept* does, in Sect. 3.2). The class file can even contain different encrypted files, one for each operating system. One could even imagine that what is dropped is source code, which will then be compiled locally. This idea is similar to the one presented in Michael Zalewski .

What makes this technique powerful is that it can be used to create a really cross-platform infection. It also can be used to encapsulate the native code so that it passes through the antivirus software on a firewall or a mail server without being detected. The antivirus can only check on the client machine if the java class file has not been stopped at the gateway level. It can also attack at different levels, for example it can infect both class files and executable files on different systems and install a malicious native code that will cooperate with the Java malware.

Future Java attacks will probably use this technique. It was almost the case with the Java.ClassLoader trojan but from the Java point-of-view the attack was not very well designed and it was trivial to analyze and understand what was going on. Such an attack made by a team of good Java hackers and system programmers could be really hard to detect, analyze and cure.

6 Conclusion

In order to illustrate what was possible with Java viruses, two of them were created by the author. The main theme of the article is that these viruses asre interesting but could become a lot more dangerous, stealthier and harder to detect and eradicate if some of the presented techniques were combined. Java is slowly but steadily maturing and is now widely used on a variety of platforms but it seems that the viral risk has not been specifically addressed. With the advent of MIDlets and the development of Java applications, applets are no longer the only target for exploit and virus writers.

Without being catastrophist, people (i.e., the AV industry, software companies and users) must understand that Java viruses are real and should therefore be seriously studied so that users are really as safe as they currently think they are.

Acknowledgements Many thanks to Major Eric Filiol, Head Scientist Officer of the Virology and Cryptology Laboratory of the French Army Signals Academy for supervising my research and actively helping me in the redaction of this paper, but above all for his support and encouragements.

References

Mark LaDue's Hostile Applets Home Page, http://www.cigital.com/hostile-applets/

Lindholm T, Yellin F, The Java virtual machine specification 2nd edn. http://java.sun.com/docs/books/1/index.html

Reynaud-Plantey D (2005) Reverse engineering and Java viral analysis – virus bulletin conference, Dublin, http://www.virusbtn.com/conference/vb2005/index.xml

Nolan G, Decompiling Java, APress, USA, ISBN 1-59-059265-4.

Collberg C, A taxonomy of obfuscating transformations. http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/

Filiol E (2005) Strong cryptography armoured computer viruses forbidding code analysis: the BRADLEY virus. In: EICAR 2005 Conference Proceedings, Malta, http://papers.weburb.dk/archive/00000136/, pp 216–227

Microsoft Security Bulletin MS03-011, http://www.microsoft.com/technet/security/bulletin/MS03-011.mspx

http://jola.clover.com.au/pipermail/polonet-l/2004-April/000049.html

AUSCERT ALERT – Bogus banking email allows trojan infection for outlook users, http://www.auscert.org.au/3981

CERT Advisory CA-2000-15 netscape allows Java applets to read protected resources, http://www.cert.org/advisories/CA-2000-15.html

The Apache Jakarta Project, http://jakarta.apache.org/tomcat/

Java 2 Micro Edition (J2ME) Security Vulnerabilities, http://confer\-ence.hackinthebox.org/hitbsecconf2004/speakers.php#adam

Michael Zalewski, Writing internet worms for fun and profit, http://reactor-core.org/worms-for-fun-and-profit.html