A Primer on Java Obfuscation

Stephen Torri, Derek Sanders, and Dr. Drew Hamilton
Auburn University

Gordon Evans
Missile Defense Agency

Java is not a secure language and its increasing use puts sensitive information at risk. While the authors do not recommend Java software that involves sensitive information, the current reality is that Java is used in these applications. To address this reality, this article discusses Java obfuscation techniques.

In today's software-oriented world, software ownership frequently changes. It is difficult, if not sometimes impossible, to keep track of who has a certain piece of software at any given time. This presents a problem for those who wish to keep the software internal operations a secret. Languages such as Java, which preserves a lot of high-level information in its byte code, present a problem from the standpoint of source ownership and securing it from program de-compilation. Releasing Java classes can compromise sensitive information embedded in the software, such as a missile intercept computation. Java class files contain the byte code instructions interpreted by the Java Virtual Machine (JVM). These files are easily read by programs that can recreate a source file from the class file. Java obfuscation techniques were developed to make reverse engineering harder, but many of these techniques can be defeated.

While Java was developed to be used on embedded systems, its popularity has pushed it into the public as a mainstream language. It is the view of the authors that if the developers of a program (e.g. defense industries) do not want its source code reverse engineered, then Java should not be used. Java programs cannot be protected in any manner from reverse engineering. All protection will do is slow down a determined attacker. In this article, we describe the three major techniques of Java obfuscation used in present state-of-the-art tools.

Commercial obfuscation applications generally perform three functions to secure the Java source code. First, extra loops, jumps, or even additional classes are added to change the control flow of the program so that an attacker has extra difficulty in understanding the program. Second, Package/Class/Method/Field names are renamed so they no longer state what they are for (e.g., field named 'account_balance' is now 'b'). Finally, any text strings contained in the program are encrypted.

Obfuscation Techniques

The following sections describe the three major techniques of Java obfuscation used in present state-of-the-art tools.

Control Flow Obfuscation

Control flow obfuscation is a technique that makes use of additional code and looping it to make it difficult to understand what is going on in such a way that causes an attacker to give up or confuses a tool into producing undesired results. While this strength is a good attempt at protection, there are semi-automated tools, such as LOCO [1], that allow a human user to interpret the code to distinguish between useless code and real code. While control flow obfuscation is not foolproof, it increases the difficulty an attacker has reverse engineering a program.

Control flow obfuscation is a technique that makes use of additional code and looping it to make it difficult to understand what is going on in such a way that causes an attacker to give up or confuses a tool into producing undesired results.

Name Obfuscation

Name obfuscation is used to effectively remove any information an attacker would gain by merely reading the name of fields. For example, if the original developer used meaningful names to aid development, this would also help the attacker. By changing the names, the meaning of the code is harder to understand. This is quite similar to the problem of decompiling x86 binaries. When decompiling x86 binaries into an intermediate language, e.g. Register Transfer Language, an attacker has to figure out the contents of the accumulator register and how it is used. This can be extremely tedious but not impossible. Similarly, an attacker with a Java class file, where the names are changed to simple letters (e.g. 'b' or 'c1'), is faced with a similar challenge. The strength of this method is that it removes a very useful method of program comprehension from the hands of an attacker. However, its weakness is that a human using an interactive deobfuscation environment, possibly a modified LOCO equivalent program, can discern what the variable 'b' means, and the program they use could propagate this new name 'account_balance' throughout the control flow graph where 'b' is used. Name obfuscation raises the level of difficulty in reverse-engineering, but does not make it impossible.

String Encryption

String encryption is utilized as an attempt to secure the code for a limited period of time. The more sensitive the information being protected, the stronger the encryption should be. By eliminating another source of information, obfuscation programs use this technique to increase the level of difficulty in an attempt to prevent deobfuscation. However, string encryption is almost useless since the key for decryption is contained inside the program file unless using an external key. It has been shown that attackers have already discovered how to decrypt these strings [2], rendering this obfuscation technique almost useless. Encryption is useful only if an external key is used. This, however, presents the classic key distribution and management issue. Using an external key requires securely sharing it via some mechanism, which is outside the scope of this article.

JAVA Byte Code

Java is compiled from source files into class files containing byte codes that are later interpreted or compiled into machine code at runtime in a JVM. The Java class files present a potential security problem since simply compiling the Java source code does not do enough to secure it from being recovered. Disassembly of the Java byte code is easy to do with the tools provided by Sun Microsystems as a part of its software development kit (SDK). For example, the following is the classic *Hello World* written in Java:

```
public class Hello {
public static void main ( String[]
   args )
  {
   System.out.println("Hello
   World");
  }
}
```

This example simply prints out the string saying *Hello World* to the standard console window. Compiling this program with the javac compiler produces a Java class file called Hello.class. This file is used with the Java program to produce the desired results. The Java class file can be easily disassembled into a human readable form using the javap [3] disassembler program included in the Sun Microsystems Java Development Kit (JDK).

For example, the following is the output of Hello.class after running javap:

```
Compiled from "Hello.java"

class Hello extends
   java.lang.Object {

Hello();
Code:
0: aload_0
1: invokespecial #1; //Method
   java/lang/Object."<init>":()V
4: return

public static void
   main(java.lang.String[]);
Code:
0: getstatic #2; //Field java/
   lang/System.out:Ljava/io/
   PrintStream;
3: ldc  #3; //String Hello World
5: invokevirtual #4; //Method java/
   io/PrintStream.println:
   (Ljava/lang/String;)V
8: return
}
```

We have recovered enough information that a developer with a tool such as the Dava decompiler, (McGill University's Java decompiler) included in the Java optimization framework called Soot [4], can quickly obtain the original source code seen in the following:

```
import java.io.*;

class Hello {

  Hello() {
     super();
  }

  public static void
   main(java.lang.String[] r0) {
       System.out.println
          ("Hello World");
  }
}
```

This example is a simple one but it illustrates the point. It can be seen that by merely compiling a Java application with the Sun JDK will not offer any protection against decompiling the program. This is why developers use obfuscation in order to get some level of protection against reverse engineering. Obfuscation makes it harder, but not impossible, to reverse engineer the code.

JAVA Obfuscation

Obfuscation works by confusing the flow of the source code so it is difficult to recover the intent of it. However, in order to effectively show how obfuscation works, a complex example is needed. The following code is a function that takes an integer value from the command line as an argument and reports back the list of Fibonacci[1] numbers. For example, running the command *java Fibonacci 5* will give back the calculated Fibonacci number for 5, 4, 3, and so on.

```
public class Fibonacci {

public int calculate (int n)
  {
     int output = 0;

     if (n > 1)
     {
       output = calculate (n - 1)
          + calculate (n - 2);
     }
     else
     {
        output = n;
     }
     return output;
  }
public static void main ( String[]
   inc )
   {
     if (inc.length > 0)
    {
       Fibonacci f_ref = new
          Fibonacci();
       int n =
          Integer.parseInt(inc[0]);
       while ( n != -1 )
   {
     System.out.println
       ("Calculated fibonacci
        number: " +
        f_ref.calculate ( n ) );
     n-;
   }
   }
   return;
   }
}
```

After using javap, the byte code prior to obfuscation is shown in the following:

```
public int calculate(int);
Code:
0: iconst_0
1: istore_2
2: iload_1
3: iconst_1
4: if_icmple 26
7: aload_0
8: iload_1
9: iconst_1
10: isub
11: invokevirtual #2; //Method
    calculate:(I)I
14: aload_0
15: iload_1
16: iconst_2
17: isub
18: invokevirtual #2; //Method
    calculate:(I)I
21: iadd
22: istore_2
23: goto 28
26: iload_1
27: istore_2
28: iload_2
29: ireturn
```

The commercially available obfuscation program called Zelix Klassmaster is used to obscure the names of classes, methods, and variables; encrypt any strings; and complicate the control flow. Though the nature of the program is hidden and obscured, the byte code is still easy to read. The important blocks of obfuscated byte code are explained in the following:

```
public int a(int);
```

```
Code:
0: getstatic #56; //Field A:Z
3: istore_3
```

The previous lines are loading the value of a static variable from the class A, called Z, onto the stack. The value is stored into the third local variable (var_3 = A:Z).

```
4: iconst_0
5: istore_2
```

These instructions set the second local variable to zero (var_2 = 0).

```
6: iload_1
```

This instruction loads the value of function parameter 'n' onto the stack.

```
7: iload_3
8: ifne 50
```

These instructions are checking to see if var_3 (the third local variable) is not equal to zero. If the statement returns true then it will jump to label #50, otherwise it continues to label #11. Though not seen here, at label #50 the variable 'output' is set to the value of variable 'n' and returned.

```
11:     iconst_1
12:     if_icmple 49
```

At this point, the constant integer value of '1' is loaded onto the stack, which is used to compare the value of the previous stack entry 'n' to 1. If 'n' is less than 1 then it will jump to label #49, otherwise it continues to label #15. Label #49 is not shown, but its instruction sets the variable 'output' equal to the value of variable 'n' and is returned. The two checks at lines 7-8 and 11-12 that were performed are different from the original check in the unobfuscated code to see if variable 'n' was greater than 1. The obfuscation program has altered the control flow in an attempt to obscure the nature of the function.

```
15: aload_0
16: iload_1
17: iconst_1
18: isub
19: invokevirtual #2; //Method
    a:(I)I
```

The function a:(I)I is the original function called calculate (int n) that returns an integer result. This byte code loads an object reference to the variable n, the value of variable n and a constant integer value of '1' onto the stack. It then calculates n-1 and places the result on the stack. The call to the function a:(I) with the results is the last step. This is equivalent to the function call of 'calculate (|n – 1|).'

```
22: aload_0
23: iload_1
24: iconst_2
25: isub
26: invokevirtual #2; //Method
    a:(I)I
```

These instructions are similar to the description above, except the function call is equivalent to 'calculate (n – 2).'

```
29: iadd
30: istore_2
```

At this point the results of 'calculate (n – 1)' and 'calculate (n – 2)' are taken from the stack, added together and the result is placed back on the stack. This is similar to 'calculate (n – 1) + calculate (n – 2)'. The results are stored in var_2.

```
31: iload_3
32: ifeq 51
35: getstatic #58; //Field z:Z
38: ifeq 45
41: iconst_0
42: goto 46
45: iconst_1
46: putstatic #58; //Field z:Z
```

Shown here is a reference to the variable Z from class z. However, notice that the original program did not contain a second class, but the obfuscator has added it to obscure the meaning. Labels #31-32 compare the value of var_3 to zero. If var_3 is equal to zero then the value of var_2 (original variable called 'output') is returned, otherwise, the comparison of the variable 'Z' from the class 'z' is compared to zero. If 'Z' is equal to zero then the value of 'Z' is set to 1, otherwise zero. These instructions are inserted by the obfuscator as *do nothing* statements to enhance the security and complicate deobfuscation forcing additional work to obtain the original code.

```
49: iload_1
50: istore_2
51: iload_2
52: ireturn
```

Finally, the results of the function call are returned to the original caller.

Even with obfuscation, anyone with access to the Java class files has access to the byte code and hence is capable of reversing the obfuscation process. The LOCO project, which is designed to aide a security analyst in understanding obfuscated code, could be used for this purpose. While the project is designed to look at instructions on an x86 architecture, a similar project designed for Java byte code would be much simpler to implement. This is due to the fact that the number of instructions that are represented by Java byte code is considerably less than the number of instructions for the x86 architecture. The weaknesses of obfuscation as shown with these simple examples illustrate the need for better protection against reverse engineering. In addition, the impact of obfuscation has on the performance of the software must also be analyzed and evaluated for acceptability. While many, if not most, Java developers do not read Java byte code, a determined adversary can and will.

Cost of Obfuscation

In order to effectively discuss obfuscation, the impact of obfuscation on performance with normal operations can not be ignored. Low [5] states that obfuscation should not alter the behavior of the program, which is shown next:

Obfuscating Transformation
Let $P \xrightarrow{T} P'$ be a transformation of a source program P into a target program P'.

$P \xrightarrow{T} P'$ is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, in order for $P \xrightarrow{T} P'$ to be a legal obfuscating transformation the following condition must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

The authors believe that changes to the program's control flow and the use of string encryption will inadvertently affect software performance. The degree of the impact depends on the control flow obfuscation method and encryption algorithm used. The effect of name obfuscation does not impact the run-time performance of the system. To better understand the impact of obfuscation, it must be shown in terms of runtime in a formal manner.

Control Flow Obfuscation

Intuitively, obfuscating the control flow of a Java program should incur some performance cost as it is interpreted. Definition

1 defines a performance measure for control flow obfuscation delay.

Definition 1: Control Flow Obfuscation Delay

Let $T_{ocf} = T_{cf} + \alpha$ be an equation showing the effects of obfuscation T_{ocf} on original system performance T_{cf} by time delay of control flow obfuscation α. If $T_{ocf} \leq T_{cf}$, then the obfuscation has either improved the original performance of the program or, at a minimum, met the original performance. More accurately, the equation is $T_{ocf} = T_{cf} + \alpha$, where $\alpha \leq 0$. Alternatively, if α is greater than zero, then the obfuscation has had a negative effect on system performance.

An embedded system may have hard real-time constraints which restrict how much additional delay is allowed. By real-time we refer to systems which will fail if the executing software should miss a deadline. The impact of obfuscation on the execution of the program would need to be measured – α in the equation above – to determine if it is at an acceptable level that does not degrade the system performance or user experience. That is, if $T_{ocf} > T_L$, where T_L is a limit of a real-time deadline or acceptable delay, then control-flow obfuscation may produce more harm than good.

String Encryption

String encryption on the other hand will definitely not have an obfuscation effect of zero. In the programs evaluated in [2], three of them utilized string encryption. The key used was stored in the program file along with the decryption code. The encrypted strings were either kept in the program's class files or had extra files included in the Java jar file.

The time delay caused by decryption depends on the encryption algorithm, key length, and the plain text. The original program had to access the location in memory, where the original string was, and return it to the place in the program it was used ($\delta = T_r$, where T_r is the retrieval time). Compare this to the time it takes to retrieve the encrypted string, perform the decryption algorithm, and return the plain text string ($\delta = T_{er} + T_d + T_{pr}$) where T_{er} is the time to retrieve the encrypted string, T_d is the decryption time, and T_{pr} is the time to return the string to the requester. Therefore, the time required to process and return the encrypted string should be greater than that of a non-encrypted string.

Definition 2: Encrypted String Obfuscation Delay

Let $T_{es'} = T_{ps} + \delta$ show the effect of using encrypted strings, T_{es}, on system performance using plain strings, T_{ps}, by the time delay for encrypted string decryption, δ. Then the time delay of decryption should never be zero ($\delta > 0$), therefore, $T_{es} \neq T_{ps}$, since the act of decryption is not an act that cannot be simply dismissed as some that can be ignored. Some amount of time would be required so it is more accurate to say $T_{es'} = T_{ps} + \delta$ where $\delta > 0$. The same restriction as described in Definition 1 applies. If $T_{es} > T_L$, where T_L is a limit of a real-time deadline or acceptable delay, then the time for decryption of the encrypted strings is considered a hindrance to acceptable program operations.

Combined Effects of Control Flow Obfuscation and String Encryption

The total performance impact of obfuscation can be determined by combining Definitions 1 and 2.

Definition 3: Performance Effect of Obfuscation

Let $T' = T + \alpha + \delta$ show the effect of both control flow (α) and string decryption (δ) have on the original system performance. It is important to consider both effects on performance since it is important to not rely solely on one effect for the protection of a program. Three effects shown will have an effect on security as well as an impact on performance.

Test Results

Four preliminary tests were conducted to calculate the performance cost of various methods of obfuscation. The tests were conducted on a 3GHz Pentium 4 system running Fedora Core 6 system using Java 1.6 to compile the program, Zelix Klassmaster 5.0 trial version obfuscator, and GNU Compiler Collection 4.1.1 20070105 (Red Hat 4.1.1-51) to compile the driver. A C++ driver program was created to run the target Java class file as the 'root' user on the system for 50 times and calculate the average number of central processing unit clock cycles it took to execute the target class file. The results for the tests can be seen in Table 1.

The tests show that even for a simple example the control flow obfuscation and the string encryption has some impact on the performance of the system. None of the obfuscation methods improved the performance of the target application. The impact of obfuscation must be analyzed as a part of development in order to measure the impact on system performance and user experience. Further testing and refinement of these metrics will provide a means for program managers to evaluate the performance costs of the many different Java obfuscators on the market (and in the public domain.)

Table 1: *Obfuscation Tests*

Test	Time (cpu clock cycles)	Percentage difference
Unobfuscated Fibonacci	2.7069×10^8	0%
Fibonacci program with aggressive control flow obfuscation	2.71142×10^8	+0.17%
Fibonacci program with flow obfuscation string encryption	2.71478×10^8	+0.29%
Fibonacci program with aggressive control flow obfuscation and flow obfuscation string encryption[2]	2.71356×10^8	+0.24%

Conclusion

Obfuscation is a method (albeit imperfect) to protect the intellectual property rights of its creators. Obfuscation could also be thought of as a method of protection against reverse engineering by making it difficult for a hacker to obtain a high-level representation of Java source code in order to make changes. Obfuscation does not provide any sort of run-time protection like watermarking or calculated checksums at periodic locations.

Organizations need to consider strongly what information is being released when a piece of software is distributed. *It cannot be assumed that information hard-coded into a program will not be retrieved.* This is of considerable importance when evaluating software for release through foreign military sales or other coalition partner arrangements.

For those looking to secure their software, there are professional tools available

that make claims of high dependability. Many companies offer tools for both Java obfuscation as well as .NET obfuscation. Additional claims of these tools are that they reduce package size and increase efficiency. Evaluation of these claims is on our list of future work.

It is generally agreed that Java can be reverse engineered. Obfuscation only slows you down, but obfuscation also increases the costs of reverse engineering sufficiently to deter many economic motives for reverse engineering. Anyone who dismisses obfuscation has probably not tried to reverse engineer non-trivial programs. Reverse engineering of militarily sensitive software is not constrained by the same economics as commercial software.

Why is Java used in defense software? Reducing development costs is one reason. Often, after the software has been delivered, there are compelling reasons to make the software available under foreign military sales. It is then too late to observe that Java should not be used and translating millions of lines of code of Java into something else is not a feasible option. What do you do? Obfuscation certainly does not solve this problem, but it is an option that government program managers acquiring software-intensive systems should be aware of as well as the larger issue of programming language selection in terms of software requirements and design.◆

References

1. "LOCO: An Interactive Code (De)Obfuscation Tool." ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation, 2006.
2. "Cracking String Encryption in Java Obfuscated Bytecode." Subere 2006 <www.milw0rm.com/papers/117>.
3. "The Java Class File Disassembler." Java Sun <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javap.html>.
4. Miecznikowski, J., and L. Hendren. "Decompiling Java Using Staged Encapsulation." Proc. of the 8th Conference on Reverse Engineering, 2001.
5. Low, D. "Java Control Flow Obfuscation." Thesis. University of Auckland, 1998 <www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/>.

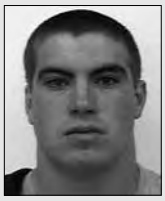
Notes

1. The Fibonacci numbers are the sequence of numbers $\{F_n\}_{n=1}^{\infty}$ defined by the *linear recurrence equation* $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. As a result of the definition, it is conventional to define $F_0 = 0$. (Wolfram Math Word <http://mathworld.wolfram.com/FibonacciNumber.html>).
2. The average time of aggressive control flow obfuscation and string encryption is most likely due to the fact that the control flow obfuscation has be optimized in some manner.

About the Authors

Stephen Torri is a doctoral candidate at Auburn University. He has a bachelor of science in accounting, finance, and computer science from Lancaster University in the United Kingdom and a master of science in computer science from Washington University in Saint Louis. Torri was an electronics technician in the U.S. Navy's Nuclear Power Program as a reactor operator aboard the USS Carl Vinson.

Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, AL 36849
Phone: (334) 844-7002
Fax: (334) 844-6329
E-mail: torrisa@auburn.edu



Derek Sanders is a graduate student studying Data Networks and Information Assurance with Auburn University. He is currently pursuing his masters degree in software engineering. Sanders' research interests include the medium access control layer for wireless communication, computer and network security, and a wide selection of issues related to securing wireless communications.

Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, AL 36849
Phone: (334) 844-7002
Fax: (334) 844-6329
E-mail: sandede@auburn.edu

John A. "Drew" Hamilton Jr., Ph.D., is an associate professor of computer science and software engineering at Auburn University and director of its information assurance laboratory. Prior to his retirement from the U.S. Army, he served as the first director of the Joint Forces Program Office and on the staff and faculty of the U.S. Military Academy, as well as chief of the Ada Joint Program Office. Hamilton has a bachelor's degree in journalism from Texas Tech University, masters degrees in systems management from the University of Southern California and in computer science from Vanderbilt University, as well as a doctorate in computer science from Texas A&M University.

Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, AL 36849
Phone: (334) 844-6360
Fax: (334) 844-6329
E-mail: hamilton@auburn.edu

Gordon Evans retired from the U.S. Army in 1992 as a Lieutenant Colonel. During his military service, he served in multiple field artillery, military intelligence, and overseas assignments. Since his retirement, Evans has worked as an on-site consultant to the Missile Defense Agency (MDA) where his areas of concentrations include systems engineering, command and control, modeling and simulations, international programs, and export control and technology transfers. He has been the lead MDA designer and investigator for its modeling and simulation vulnerability assessment program.

MDA
7100 Defense Pentagon
ATTN: MDA/BC
Washington, D.C. 20301-7100
Phone: (703) 697-4582
Fax: (703) 695-6133
E-mail: gordon.evans.ctr@mda.mil