Decompiling Java

GODFREY NOLAN

Apress®

Decompiling Java
Copyright © 2004 by Godfrey Nolan

Lead Editor: Gary Cornell

Technical Reviewer: John Zukowski

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Tracy Brown Collins

Copy Edit Manager: Nicole LeClerc

Copy Editor: Rebecca Rider

Production Manager: Kari Brooks

Production Editor: Katie Stence

Proofreader: Linda Seifert

Compositor and Artist: Kinetic Publishing Services, LLC

Indexer: Rebecca Plunkett

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Library of Congress Cataloging-in-Publication Data
Nolan, Godfrey.
Decompiling Java / Godfrey Nolan.
p. cm.
Includes index.
ISBN 1-59059-265-4 (alk. paper)
1. Java (Computer program language) I. Title.

QA76.73.J38N65 2004
005.13'3—dc22 2004014051

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 4

Protecting Your Source: Strategies for Defeating Decompilers

Now that we've addressed the problem, you're probably wondering if there is any way you can protect your code. If you're at the point of asking why you should be producing Java applets or applications that can be easily circumvented, then this is the chapter for you.

In the previous chapters, you've seen that, for a number of reasons, Java classfiles contain an unusually large amount of symbolic information. Classfiles that haven't been protected in some way return code that is almost identical to the original—except, of course, that it completely lacks any programmer comments. This chapter looks at the steps you can take to limit the amount of information in a classfile and hopefully, make the decompiler's job as difficult as possible.

Decompilation is a nasty problem from a developer's perspective. What is the point of trying to license an applet or even produce demo copies that someone can decompile or disassemble to circumvent all your protections? It seems that it is almost impossible to build in any failsafe protection mechanisms. So in this chapter, I'll introduce you to the current protection schemes as well as touch on what might be coming around the corner.

Readers of this book will probably have a foot in one of two different camps: as programmers, they may be interested in understanding how others achieve interesting effects, but from a business point of view, nobody will want someone else relabeling their code and selling it to third parties as if it was their own. Worse still, under certain circumstances, decompiling Java code can allow someone to attack other parts of your systems. A classic example of this type of problem or method of attack is when database logins and passwords are exposed after some JDBC code is decompiled. Worse still, Trojan horses can be placed in legitimate applets, which are subsequently recompiled and passed off as the original while they collect information such as logins and passwords for possible later use.

> **NOTE** *As a quick aside, you may have a much simpler reason for wanting to protect your code. Maybe decompilation should be a standard tool for the CTO or even the CEO. When this book was in its infancy, I was in the habit of decompiling everything in sight to see different styles and techniques. However, a number of times I came across code that really should never have seen the light of day. My favorite example was a software company that was about to go public that had a method in their program called* `updateS**t();`.

What would be ideal would be a black box application that would take a class-file as input, and output an equivalent protected version. Unfortunately, as of yet, nothing out there can offer complete protection. It probably helps if I defined exactly what I am aiming to do when I talk about protecting your source. Perhaps the following quote will help.

> *[We want] to protect the code by making reverse engineering so technically difficult that it becomes impossible or at the very least economically inviable.*
>
> *–Collberg, Thomborson, and Law*[1]

It is difficult to define criteria for evaluating each strategy. I'll try to measure just how effective each tool or technique is using the following three criteria:

- Just how confused is the decompiler (potency)?
- Can it repel all attempts at decompilation (resilience)?
- What is the application overhead (cost)?

You're looking for the potency, or strength, of each technique; you also need to measure how resilient the strategy is against some means of automatically removing the protection; and finally, you need to know what the overhead, or cost, associated with protecting the source is. If the performance of the code is badly degraded, then that's probably going to make the cost too high, or if you convert your code into server-side code using web services, for example, then that's going to incur a much greater ongoing cost than a standalone application.

Let's look at what strategies you can apply to reach your goal—to learn what obfuscators and other tools are available on the market and how effective they are at protecting your code. I'll also talk about how to measure their effectiveness and also touch on what to look out for in the future.

In the first chapter, you already saw the examples of how code can be protected in the judicial system. The following is an *almost* complete list of ways of protecting your Java source code before it gets to that stage.

---

1. "A Taxonomy of Obfuscating Transformations" `http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/`

- Using compilation flags and third-party compilers
- Writing two versions of the applet or application
- Employing obfuscation
- Applying web services and server-side execution
- Using encryption
- Using digital rights management
- Fingerprinting your code
- Selling the source code
- Using native methods

Compilation Flags

Two different types of compilation flags appear to have an impact on the generated bytecode. These are as follows:

```
javac –g:none|{source,lines,vars)
javac –O
```

The `-g` flag is responsible for generating all debugging information. This tells javac to add line numbers (lines option) and local variable names (vars option), which means that the classfile and constant pool is that much bigger with the added variables and the line number attributes. In Chapter 2, you saw how this allows you to map the bytecode onto the original source code. If you use this option, you can store the name of the original source file in the attributes (source option). Compiling with `–g:none` will keep lines, vars, and the source file attribute information out of your classfile. You can use `HelloWorld.java` in Listing 4-1 to see what effect compilation flags have on the bytecode.

Listing 4-1. HelloWorld.java

```
import java.applet.Applet;
import java.awt.Graphics;
import java.net.InetAddress;
import java.net.UnknownHostException;
```

```
public class HelloWorld extends Applet {

    public String getLocalHostName() {
        try {
            InetAddress address = InetAddress.getLocalHost();
            return address.getHostName();
        }
        catch (UnknownHostException e) {
           return "Not known";
        }
    }
    public void paint(Graphics g) {
    public void paint(Graphics g) {
                String s = "Hello ";
                int w = 50;
                int h = 25;

                g.drawString(s + getLocalHostName() + "!", w,h);
    }
}
```

I compiled the `HelloWorld.java` example in Listing 4-1 using `-g:`*`source,line,vars`* and then output the classfile's information using `javap -c -l` in Listing 4-2. If you compile the source using `-g` with no options, the compiler drops the local variables section but still includes line number and source file attributes. Compiling with `-g:none` flags will further remove the line number and source file information.

Listing 4-2. HelloWorld.classfile

```
Compiled from HelloWorld.java
public class HelloWorld extends java.applet.Applet {
    public HelloWorld();
    public java.lang.String getLocalHostName();
    public void paint(java.awt.Graphics);
}

Method HelloWorld()
   0 aload_0
   1 invokespecial #1 <Method java.applet.Applet()>
   4 return

Line numbers for method HelloWorld()
   line 6: 0
```

```
Local variables for method HelloWorld()
   HelloWorld this  pc=0, length=5, slot=0

Method java.lang.String getLocalHostName()
   0 invokestatic #2 <Method java.net.InetAddress getLocalHost()>
   3 astore_1
   4 aload_1
   5 invokevirtual #3 <Method java.lang.String getHostName()>
   8 areturn
   9 astore_1
  10 ldc #5 <String "Not known">
  12 areturn
Exception table:
   from   to  target type
     0     9     9   <Class java.net.UnknownHostException>

Line numbers for method java.lang.String getLocalHostName()
   line 10: 0
   line 11: 4
   line 14: 9

Local variables for method java.lang.String getLocalHostName()
   HelloWorld this  pc=0, length=13, slot=0
   java.net.InetAddress address  pc=4, length=5, slot=1
   java.net.UnknownHostException e  pc=10, length=3, slot=1

Method void paint(java.awt.Graphics)
   0 ldc #6 <String "Hello, ">
   2 astore_2
   3 bipush 50
   5 istore_3
   6 bipush 25
   8 istore 4
  10 aload_1
  11 new #7 <Class java.lang.StringBuffer>
  14 dup
  15 invokespecial #8 <Method java.lang.StringBuffer()>
  18 aload_2
  19 invokevirtual #9 <Method java.lang.StringBuffer append(java.lang.String)>
  22 aload_0
  23 invokevirtual #10 <Method java.lang.String getLocalHostName()>
  26 invokevirtual #9 <Method java.lang.StringBuffer append(java.lang.String)>
  29 ldc #11 <String "!">
  31 invokevirtual #9 <Method java.lang.StringBuffer append(java.lang.String)>
```

```
  34 invokevirtual #12 <Method java.lang.String toString()>
  37 iload_3
  38 iload 4
  40 invokevirtual #13 <Method void drawString(java.lang.String, int, int)>
  43 return

Line numbers for method void paint(java.awt.Graphics)
   line 18: 0
   line 19: 3
   line 20: 6
   line 22: 10
   line 23: 43

Local variables for method void paint(java.awt.Graphics)
   HelloWorld this  pc=0, length=44, slot=0
   java.awt.Graphics g  pc=0, length=44, slot=1
   java.lang.String s  pc=3, length=40, slot=2
   int w  pc=6, length=37, slot=3
   int h  pc=10, length=33, slot=4
```

If you decompile the code using Jad, you can see, in Listing 4-3, that compiling with the `-g:none` option is about the same as running the classfile through a very primitive renaming obfuscator, but the code is still very much intact.

Listing 4-3. Decompiled Verison of HelloWorld.java

```
// Decompiled by Jad v1.5.8e2. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://kpdus.tripod.com/jad.html
// Decompiler options: packimports(3)

import java.applet.Applet;
import java.awt.Graphics;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class HelloWorld extends Applet
{

    public HelloWorld()
    {
    }

    public String getLocalHostName()
    {
        try
```

```
        {
            InetAddress inetaddress = InetAddress.getLocalHost();
            return inetaddress.getHostName();
        }
        catch(UnknownHostException unknownhostexception)
        {
            return "Not known";
        }
    }

    public void paint(Graphics g)
    {
        String s = "Hello, ";
        byte byte0 = 50;
        byte byte1 = 25;
        g.drawString(s + getLocalHostName() + "!", byte0, byte1);
    }
}
```

Once, the `-O`, or optimization flag, did perform some rudimentary optimizations, but since Java 2 SDK, version 1.2 was introduced, this flag does not seem to have performed any optimization. In earlier versions of the JDK the optimization flag inlined `static`, `final`, and `private` methods, which meant they executed marginally faster and could handle slightly larger classfiles. One can only suspect that JIT compilers such as Hotspot are much more efficient at optimizing bytecode at runtime, so you don't really need an optimization flag at compile time. The `-O` flag now only exists for backward compatibility reasons. No doubt plenty of makefiles and Ant scripts would crash if the `-O` option was pulled from the next version of javac.

If you do a lot of debugging and like the `-g` flag, then it will help protect the code a little if you use `-g:none` because all the variable name and line number information will be lost. So change the flag before doing a final build, and if you are using a third-party Java IDE, make sure that the default compilation flag is set to `-g:none`. As you can see from Listings 4-1 and 4-3, I'm not exactly talking about huge impediments to the decompilation process, but why give the decompiler any more information than necessary?

Although other third-party Java compilers may be different, I've found that IBM's Jikes compiler performs almost identically to Sun's javac. So, currently, it looks like the compilation flags are not going to get very far in protecting your source.

Finally, a small word of warning to developers—method names and variables are very visible in Java. The Reflection API will return all the methods in a classfile, so please do not choose embarrassing names for methods or variables in Java. I'm sure vulgar method names are pretty common in other languages, but such names are just so much easier to recover in Java, and therefore, are much more likely to

embarrass you or your company in the long run. Enough preaching, but if you do tend to use strange and unusual names, then whatever you do, obfuscate with one of the better obfuscators. Nothing is better at hurting a business proposal or generating adverse publicity before an Initial Public Offering (IPO) than some bad publicity from a badly chosen method or variable name—assuming the days of the software IPO return again someday.

Writing Two Versions of the Applet or Application

Standard marketing practice in the software industry, especially on the Web, is to allow users to download a fully functional evaluation copy of the software that stops working after a certain period of time or number of uses. The theory behind this try-as-you-buy system is that after the allotted time, say 30 days, the user has become so accustomed to your program that they happily pay for a full version.

However, most software developers realize that these fully functional evaluation programs are a double-edged sword. They show the full functionality of the program but are often very difficult to protect no matter what language we're talking about. In Chapter 3, you saw how handy hexadecimal editors are at ripping through licensing schemes whether they are written in C++, Visual Basic, or indeed Java.

Many different types of protection schemes can be employed, but in the world of Java, you only have one very simple protection tool:

```
if boolean = true
        execute
else
        exit
```

These types of schemes have been cracked since the first time they appeared in VB shareware. The protection is simply modified by flipping a portion of code in the hexadecimal editor to the following:

```
if boolean = false
        execute
else
        exit
```

A number of software packages claim to be able to protect and license your software. My advice is to treat this type of program very skeptically unless it is obvious that the software is offering a new angle to this problem. Ask for an example of a protected program and use your favorite decompiler and disassemblers to see if it is truly protected.

The simplest way to steal an applet doesn't even require a decompiler or disassembler. Assuming you've viewed the applet in your browser, open the HTML to find the name of the applet and then copy it from your cache onto

your web server. Create a new HTML page and copy everything with the original <applet></applet> tags into your HTML.

When Java was in its infancy—to protect from this dastardly behavior—applets used all sort of `getDate()`, `getDocumentBase()`, `getCodeBase()`, `getHost()`, and `getLocalHost()` combinations to try to make sure that your applet was only downloaded from a licensed server, but these are exceptionally simple protection schemes and are trivial to bypass, even without a decompiler.

`getDocumentBase()` returns the host that served the web page containing the applet, and `getCodeBase()` returns the address of the applet class files. So you can make sure that the web page is only one server up from your web server by writing some code similar to Listing 4-4.

Listing 4-4. Simple Protection Mechanism

```
public void init() {
        String s = urlencode(getDocumentBase().getHost());
                if((s.compareTo("www.riis.com")) == 0 ){
                        // continue
                }else{
                         System.exit(0);
                }
}
```

You can make the program execute on another web server by changing the ==0 to !=0 using a disassembler, which leaves you in the ironic position where the modified applet now runs on every web server except the original web server. You could also decompile the code, remove the offending check, and recompile it to create an unprotected applet.

Several licensing schemes extended this idea by adding public and private keys to attempt to protect your applet. JTimer from InetSoft Technology Corporation was one such licensing tool. Mark LaDue—author of HoseMocha—took their tool apart in a similar fashion to what I just showed you in Listing 4-4 in his paper "The Maginot License: Failed Approaches to Licensing Java Software Over the Internet." Personally I agree with Mark LaDue's analysis and I don't like the primitive licensing schemes that were so prevalent when Java was mostly used for writing applets.

How much better it would be if you could write a demonstration applet or application that gives the potential customer enough of a flavor of the product without giving away the goods? For instance, you could consider crippling the demo by removing all but the basic functionality while still leaving in the menu options. If that's too much, then consider using a third-party vendor such as WebEx. It allows the potential customer to see your application, but the customer never gets a chance to run it against a decompiler.

Of course, this doesn't stop anyone from decompiling a legitimate copy of the fully functional version after they've bought it, removing any licensing schemes,

and then passing it on to other third parties. But they will have to pay to get that far, and often that is enough of an impediment to hackers that they will simply look elsewhere.

> **NOTE** *A not exactly trivial alternative to this licensing approach for applets would be to create an automated robot that searches the web for applets with the same name, size, and fingerprint. The robot could then automatically send an email to the people listed in the DNS record for that domain name to ask them to remove your applet, assuming that the fingerprint matches.*

In the next section, we'll look at what obfuscators are available and what they can do to help you get over this hurdle.

Employing Obfuscation

Maybe a dozen or so different Java obfuscators have seen the light of day. Most of the earlier versions of this type of technology are now pretty difficult to find. You can still find traces of them on the Web if you look hard enough, but apart from one or two notable exceptions, Java obfuscators have mostly faded into obscurity—yet another example of the bottom falling out of the dotcom market.

This leaves you with the interesting problem of how you tell if any of the remaining handful of obfuscators are any good. Or perhaps we've lost something very useful in the original obfuscators that would have protected your code but couldn't hold on long enough when the market took a turn for the worse? You need to understand what obfuscation really means because you have no way of knowing whether one obfuscator is better than another, unless you use market demands as your deciding factor.

When obfuscation is outlawed, only outlaws will sifjdifdm wofiefiemf eifm.

—Paul Tyma, PreEmptive Solutions

In this section, you're going to look at obfuscation theory, and you'll get a little practice. To begin with, it might help if we borrow from Christian Collberg's "Taxonomy of Obfuscating Transformations" to help shed some light on where exactly we stand. In his paper, Christian splits obfuscation into three distinct areas.

- Layout obfuscation
- Control obfuscation
- Data obfuscation

Table 4-1 lists a reasonably complete set of obfuscations that I've separated into these three different types, and in some cases, further classified. You'll take a look at the more important transformations in each section, as this chapter progresses.

Table 4-1. Obfuscation Transformations (with apologies to Christian Collberg)[2]

Obfuscation Type	Classification	Transformation
Layout		Scramble identifiers
Control	Computations	Insert dead or irrelevant code
		Extend loop condition
		Reducible to non-reducible
		Add redundant operands
		Remove programming idioms
		Parallelize code
	Aggregations	Inline and outline methods
		Interleave methods
		Clone methods
		Loop transformations
	Ordering	Reorder statements
		Reorder loops
		Reorder expression
Data	Storage and encoding	Change encoding
		Split variable
		Convert static to procedural data
	Aggregation	Merge scalar variables
		Factor class

2. Some transformation types, which are particularly ineffective for Java, are omitted in this table.

Table 4-1. Obfuscation Transformations (with apologies to Christian Collberg) (continued)

Obfuscation Type	Classification	Transformation
		Insert Bogus class
		Refactor class
		Split array
		Merge arrays
		Fold array
		Flatten array
	Ordering	Reorder methods and Instance variables
		Reorder arrays

Most of the Java obfuscators you'll meet only perform layout obfuscation with some limited data and control obfuscation. This is partly due to the Java verification process throwing out any illegal bytecode syntax. The Java Verifier is very important if you write mostly applets because remote code is always verified. These days, where there are fewer and fewer applets, the main reason Java obfuscators don't feature more high-level obfuscation techniques is because the obfuscated code has to work on a variety of Java Virtual Machines (JVMs).

Although the JVM specification is pretty well defined, each JVM has its own slightly different interpretation of the specification, which leads to lots of idiosyncrasies when it comes to how a JVM will handle bytecode that can no longer be represented by Java source. JVM developers don't pay much attention to testing this type of bytecode, and your customers aren't interested in whether or not it's syntactically correct; they just want to know why it won't run on their platform.

As you look into these different areas, please remember that you'll need to employ a certain degree of tightrope walking in advanced forms of obfuscation, what I call *high-mode obfuscation,* so you need to be very careful about what these programs can do to your bytecode. The more vigorous the obfuscation, the more difficult it is to decompile, but the more likely it will fail to pass the Java Verifier or crash some obscure JVM.

The best obfuscators will perform multiple transformations without breaking the Java Verifier or any JVM. Not surprisingly, the obfuscation companies err on the side of caution, which inevitably means less protection for your source code.

Layout Obfuscations

Most obfuscators work by obscuring the variable names or scrambling the identifiers in a classfile to try and make the decompiled source code useless. As you saw in Chapter 2, this doesn't stop the bytecode from getting executed because the classfile uses pointers to the methods names and variables in the constant pool rather than the actual names.

Obfuscated code mangles the source code output by a decompiler by renaming the variables in the constant pool with automatically generated garbage variables while still leaving the code syntactically correct. In effect, it removes all clues that a programmer gives when naming variables (most good programmers will have chosen meaningful variable names). It also means that the decompiled code will require some rework before it can be recompiled.

However, most capable programmers can make their way through obfuscated code with or without the aid of hints from the variable names. With due care and attention—and perhaps the aid of a profiler to understand the program flow and maybe a disassembler to rename the variables—most obfuscated code can be changed back into something easier to handle no matter how significant the obfuscation.

Crema is the original obfuscator and was a complementary program to the oft-mentioned Mocha, written by the late Hanpeter Van Vliet. Mocha was given away free, but Crema cost somewhere around $30. To safeguard against Mocha, you had to buy Crema. It performed some rudimentary obfuscation and had one interesting side effect. It flagged class files so that Mocha refused to decompile any applets or applications that had been previously run through Crema. However, other decompilers soon came onto the market, and they were not so Crema friendly.

Early obfuscators such as JOBE[3] replaced the method names with `a,b,c,d ... z()`. Crema's identifiers were much more unintelligible, using Java-like keywords to confuse the reader, as shown in Listing 4-5. Several other obfuscators went one step further by using Unicode style names, which had the nice side effect of crashing many of the existing decompilers.

Listing 4-5. Crema-Protected Code

```
private void _mth015E(void 867 % static 931){
        void short + = 867 % static 931.openConnection();
        short +.setUseCaches(true);
        private012O0126013D = new DataInputStream(short +.getInputStream());
        if(private012O0126013D.readInt() != 0x5daa749)
                throw new Exception("Bad Pixie header");
        void do const throws = private012O0126013D.readShort();
```

3. http://www-personal.engin.umich.edu/java/unsupported/jobe/doc.html

```
        if(do const throws != 300)
                throw new Exception("Bad Pixie version " + do const throws);
        _fld015E = _mth012B();
        for = _mth012B();
        _mth012B();
        _mth012B();
        _mth012B();
        short012O0129 = _mth012B();
        _mth012B();
        _mth012B();
        _mth012B();
        _mth012B();
        void |= = _mth012B();
        _fld013D013D012Oimport = new byte[|=];
        void void = |= / 20 + 1;
        private = false;
        void = = getGraphics();
        for(void catch 11 final = 0; catch 11 final < |=;){
                void while if = |= - catch 11 final;
                if(while if > void)
                        while if = void;
                private012O0126013D.readFully(_fld013D013D012Oimport,
catch 11 final, while if);
                catch 11 final += while if;
                if(= != null){
                        const = (float)catch 11 final / (float)|=;
                        =.setColor(getForeground());
                        =.fillRect(0, size().height - 4,
 (int)(const * size().width), 4);
                }
        }
}
```

JOBE is probably more useful as an unobfuscator than as an obfuscator because of the way it renames methods and variables, getting rid of any Unicode or Java keyword names in the process—nothing wrong with a bit of lateral thinking, especially in the field of reverse engineering. Alternatively you can use something like SourceAgain's automatic variable name generation.

Most of the obfuscators we've met, such as Crema and JOBE, are much better at reducing the size of a classfile rather than protecting the source. However, there is a small twist in the tale, because PreEmptive Solutions holds a patent that breaks the link between the original source and obfuscated code and goes some way toward protecting your code.

All the methods are renamed to a, b, c, d, and so on. But unlike other programs, as many methods as possible are renamed using operator overloading wherever

possible. Overloaded methods have the same name but have different numbers of parameters, so more than one method can be renamed `a()`, as shown here:

```
getPayroll()                          becomes   a()
makeDeposit(float amount)             becomes   a(float a)
sendPayment(String dest)              becomes   a(String a)
```

The classic example from PreEmptive shows the following:

```
// Before Obfuscation

private void calcPayroll(RecordSet rs) {

    while (rs.hasMore()) {
        Employee = rs.getNext(true);
        Employee.updateSalary();
        DistributeCheck(employee);
    }
}

// After Obfuscation

private void a(a rs) {

    while (rs.a()) {
        a = rs.a(true);
        a.a();
        a(a);
    }
}
```

Giving multiple names to the different methods can be very confusing. True, the overloaded methods are difficult to understand, but they are not impossible to comprehend. They too can be renamed into something easier to read. Having said that, operator overloading has proved to be one of the best layout techniques to beat because it does break the link between the original and the obfuscated Java code.

Control Obfuscations

The concept behind control obfuscations is to confuse anyone looking at decompiled source by breaking up the control flow of the source.

Functional blocks that belong together are broken apart and functional blocks that don't belong together are intermingled to make the source much more difficult to understand.

Collberg's paper breaks down control obfuscations further into three different classifications of *computation, aggregation,* and *ordering*. You'll now look at some of the most important of these obfuscations or transformations in a little more detail.

`Computation`

If you refer back to the computation classification section of Table 4-1, you see that it can be broken down into the following transformations.

Insert Dead or Irrelevant Code

You can insert dead code or dummy code to confuse your attacker; this can include extra methods or simply a few lines of irrelevant code. If you don't want the performance of your original code affected, then add the code so that it never gets executed. But be careful, because many decompilers and even obfuscators remove code that never gets called.

Don't just limit yourself to thinking about inserting Java code; there's no reason why you can't insert irrelevant bytecode. Mark LaDue wrote a small program called HoseMocha that altered a classfile by adding a `pop` bytecode instruction at the end of every method. As far as most JVMs were concerned, this was an irrelevant instruction and was simply ignored. However Mocha couldn't handle it and crashed. No doubt if Mocha's author had survived, then it could have been easily fixed, but he didn't.

Extend Loop Condition

Obfuscate the code by making the loop conditions much more complicated. You do this by extending the loop condition with a second or third condition that doesn't do anything. It should not affect the number of times the loop is executed or decrease the performance. Try to use the bitshift or ? operator in your extended condition for some added spice.

Reducible to Nonreducible

The Holy Grail of obfuscation is to create obfuscated code that cannot be converted back into its original format. To do this, you need to break the link between the bytecode and the original Java source. The obfuscator transforms bytecode control flow from its original reducible flow to something nonreducible. Because Java bytecode is, in some ways, more expressive than Java, you can use the Java bytecode `goto` statement to help out.

Let's revisit an old computing adage, which states that using the `goto` statement is the biggest sin that can be committed by any self-righteous computer programmer. Edsger W. Dijkstra's "Go To Statement Considered Harmful" paper[4] was the beginning of this particular religious fervor. The anti–`goto` statement camp produced enough anti–`goto` command sentiment in its heyday to put it right up there with the best Usenet flame wars.[5]

Common sense tells us that it's perfectly acceptable to use the `goto` statement under certain limited circumstances. For example, you can use the `goto` statement to replace how Java uses the `break` and `continue` statements. The issue is in using `goto` to break out of a loop or having two `goto` statements operate within the same scope. You may or may not have seen it in action, but bytecode uses the `goto` statement extensively as a way to control the code flow. However, the scope of no two `goto`'s ever cross.

The Fortran statement in Listing 4-6 illustrates a `goto` statement breaking out of a control loop. One of the principal arguments against using this type of coding style is that it can make it almost impossible to model the control flow of a program and introduces an arbitrary nature into a computer program—which, almost by definition, is a recipe for disaster. At this point, we say that the control flow has become irreducible.

Listing 4-6. Breaking Out of a Control Loop Using a goto Statement

```
do 40 i = 2,n
if(dx(i).le.dmax) goto 50
dmax = dabs(dx(i))
40      continue
50      a = 1
```

As a standard programming technique, it's a very bad idea to attempt to have `goto` statements that cross scope because not only is it likely to introduce unforeseen side effects—because it's no longer possible to reduce the flow into a single flow graph—but it also makes the code unmanageable.

However, some argue that this is the perfect tool for protecting bytecode if you can assume that the person writing the protection tool to produce the illegal gotos knows what they are doing and won't introduce any nasty side effects. It certainly makes it much harder to reverse engineer because the code flow does indeed become irreducible, but it's important that any new constructs added are as similar as possible to the original.

A few of words of warning before I leave this topic: where it is almost without a doubt that a traditionally obfuscated classfile is functionally the same as its

4. `http://www.acm.org/classics/oct95/`

5. It is rumored that the hot air generated in such debates as Vi vs. Emacs, Microsoft vs. Unix, and now .NET vs. Java, would be enough to heat the town of Cwmbran in south Wales until 2010.

original counterpart, the same cannot be said of a rearranged version. A large amount of trust has to be placed in the protection tool, otherwise it will always be blamed for odd intermittent applet or application behavior.

More importantly, although current JVMs are lax about letting bytecodes through, future ones may not be so forgiving. Tools that use encryption or rearrange or generally corrupt the original bytecode might not pass bytecode verification in these stricter JVMs or might simply fail to work. After all, JVM developers were almost certainly not using irreducible bytecode as part of their test suites. There is already one example of a JIT not executing classfiles with irreducible flow modifications. Also, several defunct obfuscators were based on this technique. If possible, always test your transformed code on your target JVMs.

The other downside to this technique is that many reducible to nonreducible transformations can already be easily reversed using an automatic deobfuscator. In the end, I suspect that this Holy Grail is not going to defeat decompilation in the Java world.

Add Redundant Operands

Add extra insignificant terms to some of your basic calculations and round up the result before you use it. For example the following code prints k = 2.

```
import java.io.*;

public class redundantOperands {
         public static void main(String argv[]) {
                  int i=1;
                  int j=2;
                  int k;

                  k = i * j;
                  System.out.println("k = " + k);
         }
}
```

Add some redundant operands to the code as follows, and the result will be exactly the same because you've cast *k* to an integer before you printed it:

```
import java.io.*;

public class redundantOperands {
```

```
        public static void main(String argv[]) {
                int i = 1, j = 2;
                double x = 0.0007, y = 0.0006, k;

                k = (i * j) + (x * y);
                System.out.println(" k = " + (int)k);
        }
}
```

I should stress that using this technique throughout your code has the potential to degrade the performance of your application.

Remove Programming Idioms (or Write Sloppy Code)

Most good programmers will amass a body of knowledge over their careers and will constantly be adding to it.[6] For increased productivity, they will use the same components, methods, modules, and classes over and over again in a slightly different way each time. Like osmosis, a new language gradually evolves until everyone decides to do some things in more or less the same way. Martin Fowler's book *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999) is an excellent collection of how to take some existing code and refactor it into shape. Judging by the sales of this book, this has created a standard way of doing things in Java.

However, this type of language standardization creates a series of idioms that give the hacker way too many helpful hints, even if they can only decompile part of your code. So throw out all your programming knowledge, stop using design patterns or classes that you know have been borrowed by lots of other programmers, and *defactor* your existing code.

Writing sloppy code is easy and a heretical approach that gets under my skin and ultimately affects the performance and long-term maintenance of your code. A more difficult and, from my point of view, better alternative would be to rewrite a common Java class from the SDK and reference the renamed class from your application so that the hacker gets a little more confused and hopefully gives up.

Parallelize Code

Converting your code to threads can significantly increase its complexity. The code does not necessarily have to be thread-compatible as you can see in the HelloThread example in Listing 4-7. The flow of control has sifted from a sequential model to a quasi-parallel model with each thread being responsible for printing a different word.

---

6. Until someone forces them to become a manager.

Listing 4-7. HelloWorld Thread Example

```
import java.util.*;

public class HelloThread extends Thread
{
   private String theMessage;

   public HelloThread(String message) {
      theMessage = message;
      start();
   }

   public void run() {
      System.out.println(theMessage);
   }

   public static void main(String []args)
   {
      new HelloThread("Hello, ");
      new HelloThread("World");
   }
}
```

The downside of this approach is the programming overhead involved in making sure that the threads are timed correctly and any interprocess communication is working correctly so that the program executes as intended. The upside is that it could take significantly longer to realize that the code can be collapsed into a sequential model.

Aggregations

Aggregations as a form of obfuscation occur when certain elements of the code are folded together to make their structure less obvious. In this section, you will become familiar with aggregating methods and loops.

Inline and Outline Methods

In the "Compilation Flags" section, I mentioned that inlining methods—where every method call is replaced with the actual body of the method—is often used to optimize code because it removes the overhead of the call. In your Java code, this has the side effect of ballooning the code, often making it a much more daunting task to understand. You can also balloon the code by taking some of

the inlined methods and outlining them into a dummy method that looks like it's being called but doesn't actually do anything.

Mandate's OneClass obfuscator took this transformation to the extreme by inlining every class in an application into a single Java class. Like all early obfuscation tools, Mandate's OneClass is no longer with us.

Interleave Methods

Although it is a relatively simple task to interleave two methods, it is much more difficult to break them apart.

Listing 4-8 shows two independent methods, and in Listing 4-9, I have interleaved the code together so that it all appears to be connected. This example assumes that you want to show the balance and email the invoice, but there is no reason why it couldn't be interleaved to allow you to only email the invoice.

Listing 4-8. showBalance and emailInvoice

```
void showBalance(double customerAmount, int daysOld) {
        if(daysOld > 60) {
                printDetails(customerAmount * 1.2);
        } else {
                printDetails(customerAmount);
        }
}
void emailInvoice(int customerNumber) {
        printBanner();
        printItems(customerNumber);
        printFooter();
}
```

Listing 4-9. showBalanceEmailInvoice

```
void showBalanceEmailInvoice(double customerAmount,
                        int daysOld, int customerNumber) {
        printBanner();
        if(daysOld > 60) {
                printItems(customerNumber);
                printDetails(customerAmount * 1.2);
        } else {
                printItems(customerNumber);
                printDetails(customerAmount);
        }
        printFooter();
}
```

Clone Methods

Clone a method so that the same code but different methods are called under nearly identical circumstances. You could call one method over another based on the time of day to give the appearance that external factors exist when they really do not. Use a different style in the two methods or use it in conjunction with the *Interleave Method* transformation so that the two methods look very different but are really performing the same function.

Loop Transformations

Compiler optimizations often perform a number of loop optimizations. You can perform the same optimizations by hand or code them in your tool to obfuscate the code. Loop unrolling reduces the number of times a loop is called and loop fission converts a single loop into multiple loops. For example, if you know maxNum is divisible by 5, you can unroll the for loop as shown in Listing 4-10.

Listing 4-10. Loop Unrolling

```
// Before
for (int i = 0; i<maxNum; i++){
    sum += val[i];
}
// After
for (int i = 0; i<maxNum; i+=5){
    sum += val[i] + val[i+1] + val[i+2] + val[i+3] + val[i+4];
}
for (x=0; x < maxNum; x++){
    i[x] += j[x] + k[x];
}

for (x=0; x < maxNum; x++) i[x] += j[x];
for (x=0; x < maxNum; x++) i[x] += k[x];
```

Ordering

If you take a look at the Ordering Classification in Table 4-1, you can see that it can be broken down into the following transformations.

Reorder Statements and Expressions

Reordering statements and expressions have a very minor effect on obfuscating the code. However, there is one example where reordering the expressions at

a bytecode level can have a much more significant impact—when it once again breaks the link between bytecode and Java source.

PreEmptive Solutions uses a concept known as Transient Variable Caching (TVC) to reorder a bytecode expression. TVC is a straightforward technique that has been implemented in DashO. Say you want to swap two variables, x and y. The easiest way to accomplish this is to use a temporary variable, as shown in Listing 4-11. Otherwise you may end up with both variables containing the same value.

Listing 4-11. Variable Swapping

```
temp = x;
x = y;
y = temp;
```

This produces the bytecode in Listing 4-12 to complete the variable swap.

Listing 4-12. Variable Swapping in Bytecode

```
iload_1
istore_3
iload_2
istore_1
iload_3
istore_2
```

However, the stack behavior of the JVM means that you don't really need a temporary variable. The temporary or transient variable is cached on the stack and the stack now doubles as a memory location. You can quite happily remove the load and store operations for the temporary variable as shown in Listing 4-13.

Listing 4-13. Variable Swapping in Bytecode Using DashO's TVC

```
iload_1
iload_2
istore_1
istore_2
```

The downside to this is that many decompilers know about this trick and can quickly revert to the original code.

Reorder Loops

You can transform a loop, making it go backward (see Listing 4-14). This probably won't do much in the way of optimization, but it is one of the simpler obfuscation techniques.

Listing 4-14. Loop Reversals

```
x = 0;
while (x < maxNum){
        i[x] += j[x];
        x++;
}

x = maxNum;
while (x > 0){
        x--;
        i[x] += j[x];
}
```

Data Obfuscations

Take a look at the Data Obfuscation type in Table 4-1. You can break this down into the classifications discussed in the following sections.

Storage and Encoding

Many of the transformations you have seen so far exploit the fact that programmers write code following some standard conventions. Turn these conventions on their head and you have the basis of a good obfuscation process or tool. The more transformations you employ, the less likely it will be for anyone or any tool to understand the original source. In this section, you will see Data Obfuscations that reshape the data into less natural forms.

Changing Encoding

Collberg's paper shows simple encoding example—an integer variable *int i = 1* is transformed to $i' = x*i + y$. If you choose $x = 8$ and $y = 3$, you get the transformation shown in Listing 4-15.

Listing 4-15. Variable Obfuscations

```
int i = 1;                              int i = 11;
while (i < 1000) {                  while (i<8003) {
        val = A[i];                         val = A[(i-3)/8];
        i++;                                   i+=8;
}                                     }
```

Split Variables

Variables can also be split into two or more parts to create a further level of obfuscation. Collberg suggests a lookup table. For example, if you're trying to define the Boolean value of *a= true*, then you'd split the variable into *a1=0* and *a2=1* and make a lookup table like the one shown in Table 4-2 to convert it back into the Boolean value.

Table 4-2. Boolean Split Lookup Table

a1	a2	a
1	0	false
0	1	true

Convert Static to Procedural Data

An interesting if not very practical transformation is to hide the data by converting it from static data to procedural data. For example, the copyright information in a string could be generated programmatically within your code possibly using a combination of interleave transformation discussed earlier. The method to output the copyright notice could use a lookup table method similar to the one shown in Table 4-2, or it could work by combining the string from several different variables spread throughout the application.

Aggregation

Taking a look at the Aggregation Classification in Table 4-1. You can break this down into the following transformations.

Merge Scalar Variables

Variables can be merged together, or converted to a different base and then merged together. The variables values can be stored in a series of bits and pulled out using a variety of bitmask operators.

Class Transformations

One of my favorite transformations is to use threads to confuse the hacker who is trying to steal code. There is an overhead because threads are harder to understand, harder to get right. If someone is dumb enough to try to decompile

code instead of writing their own, then most likely they'll be scared off by lots of threads.

Sometimes, however, it just isn't practical to use threads because the overhead is just too big; the next best obfuscation is to use a series of class transformations. The complexity of a class increases with the depth of a class. Many of the transformations that we've discussed go against the programmer's natural sense of what's good and right in the world; however, if you use inheritance and interfaces to the extreme, then you'll be glad to hear that this will create deep hierarchies that the hacker will need time to understand.

You also don't have to defactor (see "Remove Programming Idioms") if you don't want to; you can refactor instead but with a twist. Normally refactoring simplifies code making it much more maintainable, but we can also refactor two similar classes into a parent class, leaving behind a buggy version of one or more of the refactored classes. You might also want to try refactoring two dissimilar classes into a parent class.

Array Transformations

Like variables, arrays can be split, merged, or interleaved into a single array, folded into multiple dimensions, or flattened into a one- or two-dimensional array. A straightforward approach is to split an array into two separate arrays, one containing even and the other odd indices of the array. A programmer who uses a two-dimensional array does so for a purpose; changing the dimension of the array will create a significant impediment in trying to understand your code.

Ordering Transformations

Ordering the data declarations will remove a lot of the pragmatic information in any decompiled code. Typically data is declared at the beginning of a method or just before it is first referenced. Spread the data declarations throughout your code, while still keeping the data elements in the appropriate scope.

Obfuscation Conclusion

The best obfuscator would use a number of the techniques that you've seen here. Like many of these transformations, you don't need to buy an obfuscator; you can add lots of these transformations yourself. The aim here is to confuse the would-be decompiler as much as possible by removing as much information as possible. You can do this programmatically using your own tools or simply as you write your code. Some of the transformations ask the developer to simulate what happens in an optimization stage of a compiler; others are simply bad coding practice designed to throw the hacker off the scent.

Let me mention a couple of caveats before I leave this section. First, remember that if you're going to obfuscate your code by using the same identifier multiple times in the constant pool, then you might want to talk to PreEmptive Solutions first, because they hold the patent on it. Second, you take your chances with any form of high-mode obfuscation because usually you won't have the luxury of insisting that your code is only run on certain specific JVMs. Finally, writing really bad code will make your code very difficult to read. Be careful that you don't throw the baby out with the bath water. Obfuscated code is hard to maintain and, depending on the transformation, could destroy the performance of your code. Be careful what transformations you apply.

Building Your Own Simple Obfuscator

I couldn't complete the section on obfuscation without showing you how to build your own obfuscator. The design is so simple it's almost primitive and should only be considered a starting point for your own design, but it is an obfuscator.

In Chapter 2, you got a relatively in-depth look inside the classfile structure. It won't hurt to go back to that chapter to remind yourself of the overall structure of a Java classfile. By the end of Chapter 2, you could create an XML dump of any Java classfile.

To obfuscate your target file, first run it through your disassembler—you'll find the ClassToXML code in the downloads area of the Apress web site. Open the XML file using your favorite editor, go to the constant pool section, and scramble some of the identifiers by changing the names to such identifiers as $, !, and =. The constant pool number is used throughout the classfile—that is, the pointer to the string rather than the string itself—so changing the string to something illegal in Java will not affect the functionality of your code.

Now all that remains is to turn the XML file back into a classfile. You'll find the complimentary code to your disassembler, XMLToClass, also available on the Apress web site. This takes the XML file and reassembles the file back into a binary classfile.

Let's take a look at an example of how you hand edit the XML in Listings 4-16 and 4-17 to demonstrate.

Listing 4-16. Before Obfuscation

```
<Tag_40>
  <Type>CONSTANT_Utf8</Type>
  <Value>Hello</Value>
</Tag_40>
<Tag_41>
  <Type>CONSTANT_Utf8</Type>
  <Value>java/applet/Applet</Value>
</Tag_41>
```

```
<Tag_42>
  <Type>CONSTANT_Utf8</Type>
  <Value>java/net/InetAddress</Value>
</Tag_42>
<Tag_43>
  <Type>CONSTANT_Utf8</Type>
  <Value>getLocalHost</Value>
</Tag_43>
```

We'll now take the original strings and convert them to dollar signs ($), as shown in Listing 4-17, but you're free to choose whatever character or series of characters you want. The only restriction is that the original string should be the same size as the obfuscated string. You also need to make sure that any public methods or fields are typically called by outside programs and are not modified.

Listing 4-17. After Obfuscation

```
<Tag_4O>
  <Type>CONSTANT_Utf8</Type>
  <Value>Hello</Value>
</Tag_4O>
<Tag_41>
  <Type>CONSTANT_Utf8</Type>
  <Value>java/applet/Applet</Value>
</Tag_41>
<Tag_42>
  <Type>CONSTANT_Utf8</Type>
  <Value>java/net/InetAddress</Value>
</Tag_42>
<Tag_43>
  <Type>CONSTANT_Utf8</Type>
  <Value>get$$$$$$$$$</Value>
</Tag_43>
```

Web Services and Server-Side Execution

Sometimes it's the simplest ideas that are the most effective. One of the simpler ideas for protecting code is to split your applet or, indeed, your application, and keep your source code on a remote server away from any prying eyes. The downloaded applet or application is then a straightforward GUI front end without any really interesting code. The server code doesn't even have to be written in Java.

> **CAUTION** *If you're already splitting your applet to access databases, then be careful about using existing two-tier (JDBC) or three-tier (dbAnywhere) architectures for your applets because SQL passwords can be decompiled along with the rest of the code.*

This approach is particularly suited to code that can be reworked as a web service so that not only can you protect your code but you can also register the web service with a third-party web services[7] server, creating another revenue stream in the process. This *might* go toward offsetting the increased server load and any new costs you might incur by taking this approach.

If this appeals to you, then you have several ways you can split the application. For fully functional applications, you might want to look to the Swing classes to create your interface and Java Web Start to get it and the correct JVM out to your customers. Server-side Java servlets can then do the real work behind the scenes. Web Start also makes it easy to sign your code. This helps prevent someone from disassembling your code and trying to hack into your back-end system by sending bogus transmissions in an attempt to uncover what's happening on the server. If you want to create a true web service application, then you'll probably want to put some time into investigating the Simple Object Access Protocol (SOAP).

However, by far the easiest way to split your application is to use XML-RPC where the client applet makes requests via an HTTP POST request in an XML format, as shown in Listing 4-18.

Listing 4-18. XML-RPC Client Method Call

```
<?xml version="1.0"?>
<methodCall>
        <methodName>getCube</methodName>
        <params>
                <param>
                        <value><int>3</int></value>
                </param>
        </params>
</methodCall>
```

On the server side, the method for calculating the cube of an integer is safely secured from prying eyes. True, you will need some extra servlet code for handling the XML-RPC wrapper for the responses, but the code is minimal, as you can see in Listing 4-19, which shows an example XML-RPC response.

7. Using UDDI, see `http://www.uddi.org` for more information.

Listing 4-19. XML-RPC Response

```
<?xml version=”1.0”?>
<methodResponse>
        <params>
                <param>
                        <value><int>27</int></value>
                </param>
        </params>
</methodResponse>
```

Unfortunately each of these approaches has the same disadvantage of creating or, at the very least, increasing the server load and applet or application execution speed so that it is probably only ideal for you under certain circumstances, like when you want to start charging for your web service.

Encryption

Throughout the ages, mankind has turned to encryption when trying to protect secret transmissions. Not surprisingly, several attempts have been made to prevent decompilation by encrypting classfiles so that nothing can read them except for the target JVM. If the classfile is encrypted until just before it gets executed, then nobody can decompile the code—or so the theory goes, anyway.

The developer first encrypts the classfiles to secure them. When the application is executed, the encrypted classfiles are loaded by a custom ClassLoader, which decrypts the classfiles just before passing them to the JVM. The standard way of encrypting anything in Java is to use the Java Cryptography Extension (JCE).

Now it turns out that creating a custom ClassLoader and decrypting the data is a relatively easy thing to do, as you can see in Listing 4-20.

Listing 4-20. Custom Class Loaders

```
Class CustomClassLoader extends ClassLoader {
        String key;
        CustomClassLoader (String key) {
                this key = key;
        }
        public Class findClass(String name) {
                byte[] b = loadClassData(name);
                return defineClass(name, b, 0, b.length);
```

```
        }
        private byte[] loadClassData(String name) {
                // load class

                //decrypt class

        }
}
```

So does it hold water? Or is it about as safe as an Enigma machine on D-Day? Well it doesn't take long to realize that this approach has a number of holes. At the very least, a compromised JVM can simply output the decrypted classfile to a file for later analysis. But there are also several places where the encrypted file is no longer encrypted and is vulnerable to attack. For example, the custom ClassLoader program can be decompiled, modified so that the decrypted file can be captured as a stream of bytecode, and recompiled so that it dumps the classfile just before it is passed to the JVM.

Other problems are related to key security because the cryptographic key needs to be part of the application so that you can decrypt the classes in the custom class loader. If the hacker can find the key, then they can decrypt your classfiles before they get into the class loader.

But perhaps the biggest problem with this approach is that J2EE application servers, such as IBM's WebSphere or BEA's WebLogic, are fundamentally based on custom class loaders making this an altogether much more difficult approach.

It is more expensive and not that practical, but it might be possible to convert the JVM and the associated encryption routines to a hardware solution. Then nobody could access your decrypted code, because the key would be hard wired into the chip.

However, this approach has two major disadvantages: first, you would destroy Java's portability in the process of creating your encrypted JVM on a chip; and second, you would create a very limited market for your software in the process. Aside from the marketing implications of this decision, the entire security of this solution is also predicated on the hardwired encryption key never falling into the wrong hands. Lots of electronic devices employ similar encryption mechanisms such as DVD players and cable TV set top boxes. And if you've ever heard of DeCSS or cable TV descramblers, you'll realize why this isn't necessarily the best approach.

Digital Rights Management

Perhaps we're approaching the problem from the wrong angle. We know that we need to keep bytecode out of the hands of the end user in order to be able to prevent decompilation. So why not secure the browser and class loader using

a trusted browser where the end user cannot access the internals of the browser—the browser cache.

Digital Rights Management (DRM) or Intellectual Property Rights (IPR) software is the future of mainstream computing whether you like it or not. Little by little, the media and the larger software companies are moving to a licensing model. The only way to control this is to use DRM to enforce the licensing restrictions.

The logic behind DRM or IPR protection schemes with respect to Java is that if you can't get at the classfile, then you cannot possibly decompile it, and to do that, you need a secure browser cache. Typically this type of technology uses a trusted browser that carefully controls its own cache and restricts access to any classfiles, HTML, and images.

Trusted browsers are typically used as a mechanism to protect data from being viewed by unauthorized users. This mechanism aims to handle the super distribution model where one user who buys a legitimate copy can pass the data—be it an image, HTML, or text file—to other users. The next user in the chain cannot view the data without contacting the original server to obtain a new key. User's rights, such as printing and the number of times the file can be viewed, can also be strictly controlled. InterTrust and Sealed Media are two examples of companies working in this arena.

It's important to note that the trusted browser should not be written in Java because otherwise it too can be decompiled, allowing access once again to the bytecode, albeit after a lot more work than what you've encountered so far. Be careful too that using a trusted browser to deploy your applet does not drastically limit what platforms you can support. And if you can't use Java, then each trusted browser needs to be ported to different operating systems to support multiple platforms.

To take this one stage further, Bruce Schneier's Cryptogram[8] recently talked about Microsoft's Palladium architecture. This is Microsoft's implementation of the Trusted Computing Platform Alliance (TCPA) specification for a trusted computer where even the administrator does not have full access to the underlying files on a PC.

For the moment, this technology is in its infancy, but expect it to grow. Similar protection schemes in the future are likely to provide the best chance of success in nailing the decompiler issue once and for all. Like the Homeland Security bill, this is a double-edged sword—sure, you protect your code, but in the process, you give up your access to your computer's operating system.

Fingerprinting Your Code

Although it does not actually protect your code, putting a digital fingerprint in your software will allow you to later prove that *you* wrote your code. Ideally, this

8. http://www.schneier.com/crypto-gram-0208.html

fingerprint—usually in the form of a copyright notice—will act like a software watermark that you can retrieve at any time, even if your original code has been through a number of changes or manipulations before it made it into someone else's Java application or applet. As I've said several times now, no sure-fire, 100-percent effective way for protecting your code exists, but sometimes that might not matter if you can recover some losses by proving that you wrote the original code.

In case you might be confused, I should point out that digitally fingerprinting your code is completely different than signing your applet or application. Signed applets don't have any effect when it comes to protecting your code. Signing an applet helps the person downloading or installing the software decide whether to trust an applet or not by looking at the digital certificate associated with the software. It is a protection mechanism for someone using your software; it allows them to certify that this application was written by XYZ Widget Corp. The user can then decide whether or not he or she trusts XYZ Widget Corp before continuing to download the applet or launching the application. A digital fingerprint, on the other hand, is typically recovered using a decoding tool that displays the original fingerprint or watermark. It helps protect the copyright of the developer, not the end user's hard drive.

Several attempts at fingerprinting try to protect the entire application using, for example, a defined coding style. More primitive types of fingerprinting encode the fingerprint into a dummy method or variable name. This method name or variable might be made up of a variety of parameters such as the date, the developer's name, the name of the application, and so on. However, this approach can create a Catch-22. If you put a dummy variable in your code and someone just happens to cut and paste the decompiled method complete with the dummy variable into his or her program, how are you going to know it's your code without decompiling their code and probably breaking the law in the process?

Having said that, most decompilers, and even some obfuscators, will strip this information because it does not take an active role as the code is interpreted or executed. So ultimately you need to be able to convince the decompiler or obfuscator that any protected method is part of the original program by invoking the dummy method or by using a fake conditional clause that will never be true so that the method will never get called. Here is an example:

```
if(false) then{
           invoke dummy method
}
```

A smart individual will be able to see a dummy method even if the decompiler cannot see that this clause will never be true, and he or she will come to the conclusion that the dummy method is probably some sort of a fingerprint. So you need to attach the fingerprint information at the method level to make it more robust.

Finally, you don't want the fingerprint to damage the functionality or performance of your application. Because you've seen that the Java Verifier often plays a significant role in determining what protection mechanisms you can apply to your code, you really need to make sure that your fingerprint does not stop your bytecode from making it through the Verifier.

Let's use the points made in the previous discussion to define the criteria for a good digital fingerprinting system.

- No dead-code dummy methods or dummy variables should be used.
- The fingerprint needs to work even if only part of the program is stolen.
- The performance of the applet or application shouldn't be affected. The end user of the program should not notice a difference between the fingerprinted and nonfingerprinted code.
- The fingerprinted code should be functionally equivalent to the original code.
- The fingerprint must be robust enough to survive a decompilation attack as well as any obfuscation tools.
- The bytecode should be syntactically correct to get past the Java Verifier.
- The classfile needs to be able to survive someone else fingerprinting the code with his or her own fingerprint.
- You also need a corresponding decoding tool to recover and view the fingerprint using, preferably, a secret key, because the fingerprint shouldn't be visible to the naked eye or to other hackers.

You shouldn't be too concerned about whether the fingerprint is highly visible or not. On the one hand, if it's both visible and robust then it's likely to scare off the casual hacker. On the other hand, the more seasoned attacker will know exactly where to attack. However, if it isn't visible, the casual hacker doesn't know the application is protected, and then there's no up front deterrent to look elsewhere.

Several fingerprinting systems out there satisfy some of our criteria. Let's take a look at one in particular—jmark from the Nara Institute of Science and Technology in Japan.[9] It seems to meet most of our criteria, except unfortunately you do need to insert a dummy method to make it work. This also means you'll need to insert a dummy method invocation if you're not going to lose the watermark to an obfuscator. This makes it much harder to automate. However, the

9. jmark can be found at `http://se.aist-nara.ac.jp/jmark/`

only serious alternative, SandMark, according to it's web site, needs a series of "annotations throughout the source."[10]

jmark uses the structure of the opcodes to encode the fingerprint and it claims to be able to recover a fingerprint when the original classfile is decompiled, subsequently recompiled, and finally run through an obfuscator. Because you're using a dummy method, it really doesn't matter if you replace `iadd` with `isub` because the bytecode will remain syntactically correct and still get through the Java Verifier.

It turns out that you can replace `iadd` with any of the following: `isub`, `imul`, `idiv`, `irem`, `iand`, `ior`, or `ixor`. And of course this applies to the other seven opcodes too. So you now have 3 bits of information into these opcodes, so let's assign 000_2 to `iadd`, 001_2 to `isub`, and continue in this manner all the way up to 111_2 `ixor`.

Now your fingerprinting tool just has to exploit this bytecode replacement concept by converting your fingerprint or copyright notice into base 2 or a string of bits. And every time you come across one of the eight opcodes in your dummy method, all you have to do is replace it with the next bit of your fingerprint. The decoding tool works by trawling through all the methods looking for the target opcodes, reassembling them into a base 2 string, and then converting it back into readable text.

So assuming that the dummy method is big enough for the copyright notice, then you have yourself a fingerprinting system that will survive both decompilation and obfuscation.

Fingerprinting Example

OK, so much for the theory, let's see if fingerprinting is indeed a practical solution. Listing 4-21 shows another simple example you want to protect.

Listing 4-21. Casting Target Class

```
public class Casting {
   public static void main(String args[]){
                for(char c=0; c < 128; c++) {
                        System.out.println("ascii " + (int)c + " character "+ c);
                }
   }
}
```

You'll begin with a simple class file that prints out a list of ASCII characters from 0 to 127. In Listing 4-22, you insert a dummy method with lots of operators and create a conditional statement that will never be true so that the dummy code doesn't get executed.

10. SandMark can be found at `http://cgi.cs.arizona.edu/~sandmark/sandmark.html`

Listing 4-22. Casting Class with a Dummy Method

```
public class Casting {
   public static void main(String args[]){
                int i=0;
                char c;
                for(c=0; c < 128; c++) {
                        System.out.println("ascii " + (int)c + " character "+ c);
                }
                if(i==(int)c) {
                         check_std(i);
                }
   }
    // dummy method
   private static void check_std(int k){
        int i, j;
        for(i = 0; i < 10 ; i++)
                for(j = 0; j < 10 ; j++) k+=i*10+j;
                         System.out.println("k = " + k);
        for(i = 0; i < 20 ; i++)
                for(j = 0; j < 30 ; j++) k+=i*3-j;
                         System.out.println("k = " + k);
        for(i = 0; i < 25 ; i++)
                for(j = 0; j < 20 ; j++) k+=i*4-j*3;
                         System.out.println("k = " + k);
    }
}
```

First compile the code using javac. In Listing 4-23, run jmark with no parameters to find its usage.

Listing 4-23. jmark Command-Line Parameters

```
jmark version 1.3.1
Copyright (C) 1997-2002 Akito Monden
Usage: jmark target_file(.class) method_number "watermark" [options]
Options: -k"...." : key phrase
          -a0...2  : algorithm (default = 0)
          -d       : disassemble
```

Remember that there is always a constructor, by default, even if you haven't created one; this means that in this instance, your target method is `method_number 3`. As a result, you then insert the copyright notice as shown in Listing 4-24.

Listing 4-24. Creating a Fingerprint

```
c:\>jmark Casting.class 3 "(C) RIIS LLC" -k "2secret4me"
#classfile: Casting.class
#method: 3
#watermark: "(C) RIIS LLC"
#key: "2secret4me"
#algorithm: 0 (default)
```

Listing 4-25 shows the format you would use to uncover the fingerprint. You simply run the companion program jdecode against the classfile.

Listing 4-25. Recovering the Fingerprint

```
c:\>jdecode Casting.class -k "2secret4me"
#classfile: Casting.class
#key: "2secret4me"
#algorithm: 0 (default)
#begin{watermark}
1        ""
2        "GQZQ"
3        "(C) RIIS LLC (C) RIIS LLC "
#end{watermark}
```

If you decompile the program using Jad, you can see how the `dummy_method` has been altered. It's changed pretty dramatically, so make sure you target the dummy method if you don't want to ruin your perfectly good code. The watermark is encoded by changing the bytecodes for `imul`, `isub`, and so on so that the original code, although still syntactically correct, is always different from the original dummy method (see Listing 4-26).

Listing 4-26. Decompiled Fingerprinted Code

```
// Decompiled by Jad v1.5.8e2. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://kpdus.tripod.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Casting.java

import java.io.PrintStream;
```

```
public class Casting
{
    public Casting()
    {
    }

    public static void main(String args[])
    {
        int i = 0;
        char c;
        for(c = '\0'; c < 128; c++)
            System.out.println("ascii " + (int)c + " character " + c);
        if(i == c)
            check_std(i);
    }

    private static void check_std(int i)
    {
        for(int j = 0; j < 10; j++)
        {
            for(int i1 = 0; i1 < 10; i1++)
                i += j * 10 + i1;
        }
        System.out.println("k = " + i);
        for(int k = 0; k < 20; k++)
        {
            for(int j1 = 0; j1 < 30; j1++)
                i += k * 3 - j1;
        }
        System.out.println("k = " + i);
        for(int l = 0; l < 25; l++)
        {
            for(int k1 = 0; k1 < 20; k1++)
                i += l * 4 - k1 * 3;
        }
        System.out.println("k = " + i);
    }
}
```

Because `Casting.java` is such a simple program, it is pretty obvious that the functionality of the new classfile didn't change because of the fingerprint or watermark, because our dummy method never gets called. However, it's not good enough to assume that is the case when you're fingerprinting real applications.

If you don't have a test suite to test the functionality of your code, then you're probably going to need to create one.

jmark or any other system isn't going to be 100-percent secure because a smart hacker or developer will be able remove the dummy methods, especially if they take the time to run the application through a debugger. Putting different dummy methods in different classes is a good strategy to increase your odds of at least one fingerprint surviving, and doing so will certainly catch the casual hacker.

Selling the Source Code

If source code is so readily accessible, then why not just sell it at a higher price? If your asking price is not too high, you could convince a would-be decompiler to pay for the code, as programmer's comments are usually very informative. The intent is to convince someone that it just doesn't make any sense to decompile, given the time and energy that it sometimes requires.

This won't be for everyone, but why not make some money on the fact that some people will decompile your code to copy it? In the process, you might gain some extra revenue by helping to discourage these otherwise illegal activities by selling the code at a marginally higher price.

Native Methods

If Java really is that difficult to protect, then why not protect your code by writing it in C++ or C? Java allows you to do this by using native methods through the Java Native Interface (JNI). This may not be to everyone's liking because of the portability issues, but it does safeguard the code within the imported object. Now there may be an argument that, for example, the compiled C++ isn't that much safer than Java, but in my opinion, if you want to protect your algorithms, then the safest place to put them is in a native method that can then be called by the rest of your Java application.

There are several versions native methods APIs. The original JDK 1.0.2 version used the Native Method Interface (NMI) that was cumbersome at best. NMI was replaced by the JNI in JDK 1.1, which was subsequently enhanced in the JDK 1.2.[11]

You can incorporate a native method into your code by declaring it as shown in Listing 4-27 in the appropriate method.

11. Microsoft also had another flavor called JDirect many moons ago for those interested in pure Java trivia.

Listing 4-27. Native Method Example

```
class JavaHelloWorld {

    // declare the native method
    public native void nativeHelloWorld();


    // load libhello.so or hello.dll
    // depending on your platform
    static {
        System.loadLibrary("hello");
    }


    // invoke the Java class and call
    // the native method Hello World
    public static void main(String[] args) {
        new JavaHelloWorld().nativeHelloWorld();
    }
}
```

Run the following command on your compiled Java classfile:

```
javah -jni JavaHelloWorld
```

This creates the header file for your native method, shown in Listing 4-28.

Listing 4-28. JNI header file

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JavaHelloWorld *

#ifndef _Included_JavaHelloWorld
#define _Included_JavaHelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JavaHelloWorld
 * Method:    nativeHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaHelloWorld_nativeHelloWorld
  (JNIEnv *, jobject);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

Now create your native method using the same function that was auto-generated in the header file:

```
#include "JavaHelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_JavaHelloWorld_nativeHelloWorld
  (JNIEnv *, jobject);
{
     printf("Hello, Native World!");
}
```

You compile using your favorite C compiler in `libhello.so` if you're on the Solaris platform or use `hello.dll` if you're in Windows. Now you're good to go. The Java classfile loads the native library and prints:

```
Hello, Native World!
```

A word of warning, don't be tempted to just put your protection mechanism in the native code because the hacker will simply comment out the check in the Java code. You'll also need to be careful about connection strings to databases because a good hexdump using a disassembler will recover the login and passwords—native methods or not.

Conclusion

The fact that the Java classfile format contains so much information makes it exceptionally difficult to protect the underlying source. Yet most software developers continue to ignore the consequences, leaving their intellectual property at some risk. What you're looking for in your obfuscator is a process that takes polynomial time to produce and exponential time to reverse.

I hope that the pretty exhaustive list of obfuscation transformations I went over in this chapter helps you approach something nearing that goal. At the very least, check out Christian Collberg's paper, which has enough information to digest for any developer who wants to get started in this area.

It seems that a JVM's bytecode is too open to interpretation to protect with strong obfuscation that will work on he different flavors of the JVM. Table 4-3 summarizes the approaches that you've seen in this chapter.

Although it does seem like an awful lot of trouble to go to when you could just write the application in native code, the portability of Java is very attractive. I believe that there are more than enough obfuscating transformations to make decompilation a very difficult process; I also think that it is possible to make the code so illegible that it will take very close to exponential time to understand the code event if it is completely decompiled. The problem is that you can't achieve that in anything close to a polynomial time frame just yet.

> **NOTE** *Perhaps it is worth noting that many of the original obfuscation tools didn't survive the dotcom implosion and that the companies have either folded or moved into other areas of specialization. So perhaps the market demands aren't even there and people are more than happy to live with the fact that people can recover the source from a classfile.*

Table 4-3. Protection Strategies Overview

Strategy	Potency	Resilience	Cost	Notes
Compilation flags	Low	Low	Low	
Writing two versions of the applet or application	High	High	Medium	
Obfuscation	Medium	Medium	Medium	Can break some JVMs
Web services and server-side execution	High	High	High	
Encryption	Low	Low	Low	
Digital Rights Management	High	High	High	
Fingerprinting your code	Low	Low	Low	Useful for legal protection
Selling the source code	Low	Low	Low	
Native methods	High	High	Low	Breaks code portability

One final word—be very careful about relying on obfuscation as your only method of protection. Remember that disassemblers can also be used to rip apart your classfile and allow someone to edit the bytecode directly. Don't forget that interactive demonstrations over the Web or seriously crippled demonstration versions of your software can also be very effective.