Cracking String Encryption in Java Obfuscated Bytecode

subere@uncon.org

November 2006

Abstract

This paper discusses in technical detail the type of String Encryption used by popular java obfuscation tools. The main motivation behind this work lies within the identification of attack vectors as well as potential entry points of a given obfuscated application. Following a brief introduction, a methodology for reverse engineering Java bytecode is presented. We focus on the encryption techniques deployed by the following obfuscators: Zelix KlassMaster (version 4.5.0), RetroGuard (version 2.2.0), JShrink (version 2.3.7) and Dash-O (version 3.2.0). Ways to crack the corresponding levels of String encryption as well as potential further attack vectors that such obfuscators might introduce are also examined. The findings of this paper show that cracking String encryption within the above obfuscators is a task which can be performed even in an automated way. In order to mitigate against the problem a brief proposal is outlined stating the need for polymorphic obfuscators that have the ability to select from an algorithm pool depending on the type of application as well as the level of obfuscation required.

1. Introduction

The practice of obfuscating intermediate platform independent code, such as Java bytecode or Microsoft Intermediate Language (MSIL now CIL[1]) code often carries an element solely dedicated to the "*encryption*"[2] of String values embedded within the code. As the encryption taking place is part of the obfuscation process it cannot affect the behaviour of the program or deviate from the original program operations. Thus, at runtime, the Virtual Machine needs to be able to reverse any such process of encryption within the obfuscated code.

This paper examines standard "*decryption*" techniques which can be applied to obfuscated Java bytecode that has been reverse engineered. As the algorithm and any keys used to encrypt the Strings are embedded within the application, a proof of concept is presented whereby the original String representations can be decrypted from the obfuscated code. In this analysis, typically the key value does get altered while the algorithm deployed stays the same. In order to thus expand on the proof of concept scenarios presented within this paper, we have to facilitate the change of keys within the obfuscated code.

Identifying string values within obfuscated code, can assist in the detection of any important parameters (such as embedded passwords), in addition to revealing the entry points that could be used as further attack vectors on the application using reverse engineering techniques.

Typical patterns involve identifying the areas within the source code where application text is being stored. Examples include interface names, error messages as well window and panel labels. Knowing which class holds within it, say, the String representation of an error message seen by user, automatically helps categorise obfuscated components and classes to user interface elements, architecture classes, data holders, etc.

Finally, String encryption can serve as a fingerprinting tool with respect to the obfuscator used. This implies that knowing the type of String encryption present could automatically narrow down the expected changes in other areas of the code. Such changes would not relate to the String encryption process but to other operations of the obfuscator.

Even though obfuscation tools offer a wide selection of options towards altering an application, narrowing down the obfuscator and having the ability to understand how it performs on particular code snippets, yields further information towards bypassing the obfuscation process for a given application.

---

[1] Within early releases of .NET, The Common Intermediate Language (CIL) was originally known as Microsoft Intermediate Language (MSIL). Due to standardization of C# and the Common Language Infrastructure (as well as the involvement of non-Microsoft organisations), the bytecode is now officially known as CIL. This is despite the Phoenix project still using the term MSIL (http://research.microsoft.com/phoenix/). Because of this legacy, CIL is still frequently referred to as MSIL, especially by long-standing users of the .NET framework.

[2] Out of respect to the field of cryptanalysis (especially in the vicinity of Cheltenham Spa), crypto terminology used within this paper will be initially presented in quotes, as no decent or complex algorithms where identified throughout this project. Obfuscator designers have never claimed to use strong crypto; some of them are completely against doing so; still, this footnote aims to serve those who might be offended by XOR operations being classified as cryptographic.

2. The process of reverse engineering Java bytecode

Java files which have been written according to the specification of the programming language are compiled to bytecode using the java compiler (javac). Unlike .NET java uses 8-bit code known as bytecode while .NET uses 16-bit code that can be labelled as wordcode.

In a "Hello World" scenario, let us consider the simple program below and investigate what the disassembling process for such a class would be:

```
package elucidate;

public class PasswordCheck {
  private static final String password = "ThisIsMyPassword";

  public static void main(String[] args) {
    if(args.length != 1) {
      System.out.println("Please supply argument as password");
      System.exit(1);
    }
    if(args[0].equals(password))
      System.out.println("Password Correct!");
    else
      System.out.println("Password Error");
  }
}
```

This program checks the single argument provided by the user against a hardcoded password; depending on the value entered it notifies the user. As any first year Computer Science student can tell you, using the java compiler on the source code files yields the corresponding class file, which can then be executed as follows:

```
$> java elucidate.PasswordCheck MyArgument
Password Error

$>
```

Creating the corresponding jar file for the class yields:

```
$> jar cvfm PasswordCheck.jar elucidate\MANIFEST.MF elucidate\PasswordCheck.class
added manifest
adding: elucidate/PasswordCheck.class(in = 634) (out= 413)(deflated 34%)

$> java -jar PasswordCheck.jar MyArgument
Password Error

$>
```

Despite the fact that any class file holds a binary representation of the corresponding compiled bytecode, Sun Microsystem's Java Development Kit (JDK) includes a class file dissassembler (javap) which offers the ability to effectively disassemble class files into syntactically valid java bytecode statements.

```
$> javap -c elucidate.PasswordCheck
Compiled from "PasswordCheck.java"
public class elucidate.PasswordCheck extends java.lang.Object{
public elucidate.PasswordCheck();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   aload_0
   1:   arraylength
   2:   iconst_1
   3:   if_icmpeq       18
   6:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   9:   ldc     #3; //String Please supply argument as password
```

```
   11:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   14:  iconst_1
   15:  invokestatic    #5; //Method java/lang/System.exit:(I)V
   18:  aload_0
   19:  iconst_0
   20:  aaload
   21:  ldc     #6; //String ThisIsMyPassword
   23:  invokevirtual   #7; //Method java/lang/String.equals:(Ljava/lang/Object;)Z
   26:  ifeq    40
   29:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  ldc     #8; //String Password Correct!
   34:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   37:  goto    48
   40:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   43:  ldc     #9; //String Password Error
   45:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   48:  return

}


$>
```

Note that in the above example, String values are included as comments within the disassembled code. Often, compiling the source code without any debug information prevents this from happening. In his book "Hacking Exposed J2EE & Java", Art Taylor describes this as a generic countermeasure that should be taken during compilation:

```
$> javac -g:none elucidate\PasswordCheck.java
```

In the case of the java compiler, using the above command to generate the class files still yields their corresponding String values when this file is disassembled.

Provided an attacker is willing to allocate enough time and resources into understanding specifications of the Java bytecode, virtually any class file can be reverse engineered into disassembled bytecode statements using the above technique.

In addition to the approach of reverse engineering Java bytecode through the process of disassembling the file into valid bytecode statements, there is also the more popular approach of attempting to obtain a version of the corresponding source code from the class file.

In order to gain meaningful results from the decompilation process correct tools must be used. Using the right decompiler can produce source code that is as almost an exact copy of the original source from which the class file was generated.

There are many decompiler tools available for java class files. In his book "Covert Java", Alex Kalinovsky offers a brief review of the following three Java decompilers, giving them a ranking of excellent or fair:

- JAD (Free for non-commercial use) / Excellent
- JODE (GNU Public license) / Excellent
- Mocha (Free) / Excellent

For the purposes of this paper, the investigation and selection of a decompilation tool is left as an exercise for the reader. Using such a tool will speed up the process of cracking the encryption on any obfuscated code, mainly due to not having to interpret virtual machine statements.

Typically, decompilers tend to not provide source code that can be recompiled without any further modifications. This is a cross derivative of how well a decompiler supports advanced features of java, such as inner classes and javax packages.

Parsing our original class file through our selected decompiler yields:

```
package elucidate;
import java.io.PrintStream;
public class PasswordCheck {
    private static final String password = "ThisIsMyPassword";
    public static void main(String[] array1)
    {
```

```
        if( array1.length != 1 )
        {
            System.out.println( "Please supply argument as password" );
            System.exit( 1 );
        }
        if( array1[0].equals( "ThisIsMyPassword" ) )
            System.out.println( "Password Correct!" );
        else
            System.out.println( "Password Error" );
    }
}
```

Thus, we have demonstrated two ways, one through disassembly and the other through decompilation of recovering the embedded password of the simple program above.

Obfuscation techniques aim to further complicate this process by breaking down compiled code into something that is less meaningful to the human programmer (bytecode), which still executes in exactly the same way through the Java Virtual Machine. As the next section shows they also use a number of techniques to further limit the exposure of any string values within the code.

3. Embedded String values and Obfuscation

For the purpose of cracking String encryption in Java, choosing a decompiler that has the ability to provide valid source code is not critical. Ultimately, we are looking for a signature pattern that provides the following information:

- The obfuscator used
- The encryption technique (algorithm)
- The value of any key used in the algorithm

The latter two points bypass the need to know what obfuscator has been used. Inversely, knowing which obfuscator was used and the encryption techniques that it uses allow us to proceed directly into the decryption phase.

The remainder of this section presents the encryption techniques used by known obfuscation tools. Again, using Alex Kalinovsky's *Covert Java* as a guide, we present the most popular obfuscators for Java as described in the book. These are:

- Zelix KlassMaster
- Retro Guard
- JShrink
- Dash-O

We will investigate each one separately in an attempt to identify the type of String Encryption used in each one.

4. Zelix KlassMaster

Zelix KlassMaster (version 4.5.0) offers three methods for encrypting String literals: Normal, Aggressive and Flow Obfuscate. The algorithm used for all three appears to be identical. Yet, the keys used change at every obfuscation attempt. This program will also trim any final static values identified and has a number of features that can be combined together. This makes obfuscated source code from Zelix much harder to be interpreted by a decompiler.

Having used the most paranoid settings within Zelix KlassMaster (trimming, deleting attributes, aggressive control flow obfuscation, etc.) we attempt to recover the value of the password in our original test program. Parsing the obfuscated file of PasswordCheck.class through the javap dissassembler yields the following string values:

```
   6:  ldc     #8; //String ,bw:)□q`i□□qv,=□\"
   14: ldc     #6; //String (km:□□N}□?□ps&,□
   22: ldc     #3; //String ,oa(-↓#w<.♀o}i?♫dq$;↕w$(-\se:-♂lv-
   30: ldc     #9; //String ,bw:)‼q`i←♫qk;
```

It is worth noting that this output is in Unicode format. Java has the ability of not only interpreting special characters such as \n, but also Unicode octal (\777) as well as Unicode hexadecimal (\FFFF).

In its current version, KlassMaster uses a simple XOR encryption algorithm to encrypt each character within a given string against a five character key. Within the obfuscated code, two methods are typically embedded inside the class containing String values. The first is the encryption method for single characters while the second is the encryption method for every String of length greater or equal to 2. This method takes an array of characters as its argument.

Having the above description of the algorithm and the encrypted strings, all that remains is to identify the key values used. For this, we review the output from parsing the obfuscated file of PasswordCheck.class through the javap dissassembler.

Typically, the key values are located after a tableswitch statement which is followed by multiple ixor operations. This represents the calls for encrypting each character of the String (or char array) with the corresponding key value:

```
   64:  tableswitch{ //0 to 3
                0: 96;
                1: 101;
                2: 105;
                3: 109;
                default: 114 }
   96:  bipush  124
   98:  goto    116
   101: iconst_3
   102: goto    116
   105: iconst_4
   106: goto    116
   109: bipush  73
   111: goto    116
   114: bipush  94
   116: ixor
   117: i2c
```

The key values in the above disassembled output are: 124 3 4 73 94

Indeed, using these values on the previously identified obfuscated Strings yields the following:

```
Found 4 Strings in file
Found 5 Keys in file
 124 3 4 73 94
Deciphering...
-Original: ,bw:)‼q`i↔‼qv,"
->Decoded: Password Correct!
-Original: (km:↨☼N}↓?☼ps&,↑
->Decoded: ThisIsMyPassword
-Original: ,oa(-↓#w<.♀o}i?♫dq$;↕w$(-\se:-♂lv-
->Decoded: Please supply argument as password
-Original: ,bw:)‼q`i←♫qk;
->Decoded: Password Error
```

The makers of Zelix KlassMaster have defended the use of a weak algorithm in their obfuscation product. This implementation may be revisited in future versions of the product.

5. JShrink

JShrink (version 2.3.7) uses a different approach to the encryption of string values. It creates a new file which stores an encrypted version of all the String values found within a specified class. This file is stored in a newly created package (directory) which also holds the corresponding class with the algorithm used to decrypt the parts of the file corresponding to a particular String.

Using the javap dissassembler on the obfuscated version of the PasswordCheck.class file reveals that there are no further String values embedded within the code.

```
Compiled from elucidate.PasswordCheck
public class elucidate.PasswordCheck extends java.lang.Object{
public elucidate.PasswordCheck();
  Code:
   0:   aload_0
```

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

public static void main(java.lang.String[]);
  Code:
   0:   aload_0
   1:   arraylength
   2:   iconst_1
   3:   if_icmpeq       20
   6:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   9:   iconst_1
   10:  invokestatic    #48; //Method I/I.I:(I)Ljava/lang/String;
   13:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   16:  iconst_1
   17:  invokestatic    #5; //Method java/lang/System.exit:(I)V
   20:  aload_0
   21:  iconst_0
   22:  aaload
   23:  bipush  62
   25:  invokestatic    #48; //Method I/I.I:(I)Ljava/lang/String;
   28:  invokevirtual   #7; //Method java/lang/String.equals:(Ljava/lang/Object;)Z
   31:  ifeq    48
   34:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   37:  bipush  79
   39:  invokestatic    #48; //Method I/I.I:(I)Ljava/lang/String;
   42:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   45:  goto    59
   48:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   51:  bipush  97
   53:  invokestatic    #48; //Method I/I.I:(I)Ljava/lang/String;
   56:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   59:  return

}
```

From the above, two things become apparent. Firstly, extensive trimming has taken place on the static final value of the password and secondly, any statement that involved a string value has been replaced by a method call within the newly created package. Method I.I.I appears to be taking an integer as input.

Parsing the class file using the decompiler provided with the obfuscator, provides the following code:

```
package elucidate;

import java.io.PrintStream;

public class PasswordCheck {

    private static final String password = "ThisIsMyPassword";

    public static void main(String[] String_1darray1)
    {
        if( String_1darray1.length != 1 )
        {
            System.out.println( I.I.I( 1 ) );
            System.exit( 1 );
        }
        if( String_1darray1[0].equals( I.I.I( 62 ) ) )
            System.out.println( I.I.I( 79 ) );
        else
            System.out.println( I.I.I( 97 ) );
    }
}
```

Thus, we need to focus our attention on the newly created package and its corresponding elements.

One of the new files within the '*I*' package has as name *I.gif*. However, this file is not an image file. By examining the contents of the file, we can see that it does not have a valid GIF header:

```
00000000h: 00 00 6F 53 6F 6E 6D 6C 6B 6A 69 68 6F 57 6F 56 ; ..oSonmlkjihoWoV
00000010h: 67 66 65 64 63 62 61 60 6E 57 6E 56 7F 7E 7D 7C ; gfedcba`nWnV□~}|
```

```
00000020h: 7B 7A 6F 6E 6D 6C 6B 6A 69 68 6F 57 6F 56 67 66 ; {zonmlkjihoWoVgf
00000030h: 65 64 63 62 61 60 6E 57 6E 56 7F 7E 7D 7C 7B 7A ; edcba`nWnV□~}|{z
00000040h: 7F 3B 07 06 1C 26 1C 22 16 3F 0E 1C 1C 18 00 1D ; □;...&."□?......
00000050h: 0B 7E 3F 0E 1C 1C 18 00 1D 0B 4F 2C 00 1D 1D 0A ; .~?.......O,....
00000060h: 0C 1B 4E 61 3F 0E 1C 1C 18 00 1D 0B 4F 2A 1D 1D ; ..Na?.......O*..
00000070h: 00                           ;.
```

Decompiling the file I.class which holds the encryption information yields:

```
public class I {

  static byte[] SDQU;
  static String[] append = new String[256];
  static int[] close = new int[256];

  public static synchronized final String I(int int1){
    int int2 = int1 & 0xFF;
    if( close[int2] != int1 ) {
      String String3;
      close[int2] = int1;
      if( int1 < 0 ) {
        int1 = int1 & 0xFFFF;
        String3 = new String( SDQU, int1, SDQU[int1 - 0x1] & 0xFF ).intern();
        append[int2] = String3;
      }
      return append[int2];
    }

    static {
      try {
        Object Object1 = new I().getClass().getResourceAsStream( "" + 'I' + '.' + 'g' +
'i' + 'f' );
        if( Object1 != null ) {
          int   int2   =   ((InputStream)   Object1).read()   <<   0x10   |   ((InputStream)
Object1).read() << 0x8 | ((InputStream) Object1).read();
          int int3;
          byte byte4;
          byte[] byte_1darray5;

          SDQU = new byte[int2];
          int3 = 0;
          byte4 = (byte) int2;
          byte_1darray5 = SDQU;
          while( int2 != 0 ) {
            int int6 = ((InputStream) Object1).read( byte_1darray5, int3, int2 );

            if( int6 == -1 )
            break;
            int2 -= int6;
            int6 += int3;
            while( int3 < int6 ) {
              byte_1darray5[int3] = (byte) (byte_1darray5[int3] ^ byte4);
              ++int3;
            }
          }
          ((InputStream) Object1).close();
        }
      }
      catch( Exception Exception7 ) { }
    }
}
```

The above represents the encryption technique used within the current version of JShrink in order to encrypt any non static final String values.

The contents of *I.gif* are read from file using a static method. Note the caution taken to split up the filename into individual characters. Furthermore, note the operations taking place within the read process that is responsible for creating the array of values used by the synchronised method '*I*'.

From an attacker's perspective, the newly created package can be treated as a 'black box'. As the encryption algorithm is provided within an accessible method, all that is needed are the integer arguments to any String function call. Understanding the above algorithm is not necessary in order to decrypt any String values that have been obfuscated by JShrink. Still, for completeness, this is included above.

In spite of offering a completely different approach to encrypting String values, JShrink introduces a vulnerability in its current method of encryption.

Any call to method I with an argument other than the integer values used, causes an exception to be thrown, as the String index appears to be out of bounds for the decryption operation performed.

If we look at the method calls in the decompiled PasswordCheck.class file, these are:

```
System.out.println(I.I(1));
System.out.println(I.I(62));
System.out.println(I.I(79));
System.out.println(I.I(97));
```

Adding the following print statements within the application:

```
System.out.println(I.I(56));
System.out.println(I.I(80));
System.out.println(I.I(62));
```

Causes the following exception to be thrown:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of
range: 160
        at java.lang.String.checkBounds(Unknown Source)
        at java.lang.String.<init>(Unknown Source)
        at I.I.I(I.java:26)
        at elucidate.RetrieveStrings.main(RetrieveStrings.java:13)
```

Fortunately, this is not a language such as C, as that would imply that any obfuscated code by this application would potentially be vulnerable to a buffer overrun. Still, the damage is done.

6. RetroGuard

RetroGuard (version 2.2.0) does not offer String encryption in the obfuscation process.

7. Dash-O

Dash-O (version 3.2.0) uses a similar approach to encrypting any embedded Strings as that of Zelix KlassMaster. Essentially, a method is invoked within the class files, taking as argument String and returning String. This method is typically called A_B_C_D and has the format seen below:

```
public static String A_B_C_D(String s)
{
    char ac[] = new char[s.length()];
    s.getChars(0, s.length(), ac, 0);
    char c = '\0';
    for(int i = 0; i < ac.length; i++)
        ac[i] = (char)(ac[i] - 1 ^ c++);

    return new String(ac);
}
```

Thus, a typical call to the method would be of the form:

```
A_B_C_D("!\"#$%&'()*+mdzb0P^");
```

Such code has the following bytecode signature:

```
280: invokestatic    #230; //Method A_B_C_D:(Ljava/lang/String;)Ljava/lang/String;
```

As it can be seen from the code above, this method performs basic XOR on each character of the input String against an incrementing character value. Despite the fact that a single algorithm is used to encrypt String values with this version of Dash-O, a number of variations on the above can be obtained, depending on the level of obfuscation (through other parameters) applied to the original code. As a result, the above method can also be found embedded within a separate class in the following form:

```
public static String A_B_C_D(String s)
{
    char ac[] = new char[s.length()];
    s.getChars(0, s.length(), ac, 0);
    char c = '\0';
    int i = 0;
    do
    {
        if(i >= ac.length)
            return new String(ac);
        ac[i] = (char)(ac[i] - 1 ^ c++);
        i++;
    } while(true);
}
```

The underlying algorithm stays the same; simply the for loop, which has a well known bytecode representation is replaced with a do-while statement using an incremental integer counter to reach the return state. As with other obfuscators, Dash-O does not perform any encryption on String literals that have been declared static and final.

8. Conclusions

Obfuscation as a process, aims to make code harder to understand when reverse-engineered. As this process should have no effect on the functionality of code, it is by definition a reversible process. What popular obfuscators aim to achieve is an increase in the amount of time required in order to reverse engineer the code.

As seen with encrypted String literals within Java bytecode, typically the encryption process (algorithm) stays the same. This not only allows for an attacker to easily decrypt any Strings, but also yields the obfuscator tool that has been used in the process.

With the increase in use of bytecode languages, obfuscator tools will require to carry a certain degree of polymorphism regarding the algorithms and processes they deploy.

Polymorphism would be defined with respect to obfuscation as the ability to select and cross-combine the algorithms of obfuscation used from a pool of available algorithms. This selection process would be dependant on the required level of security for the given application.

In order to establish a quantitative, but not definite measure, the level of security offered by the usage of an algorithm during obfuscation would be inversely proportional to the level of previous known usage of the algorithm for the similar obfuscation processes. This would yield that the pool of algorithms is not publicly available and that different operations of an obfuscator at different times produce distinct results.

Such polymorphism should allow for the random selection of particular obfuscation techniques (including String encryption) depending on the type of application, as well as the permitted level of exposure (in terms of time) to reverse engineering. Despite of this effectively going against Kerckhoffs' principle (also known as Shannon's maxim) stating that all information apart from the key should be made available to an attacker, obfuscation as a process falls more in the category of "security through obscurity" and should be treated as such. As a result the levels of obscurity introduced should change dynamically and depending on the type of source code, thus leading to polymorphic obfuscation techniques.