Fingerprinting and Cracking Java Obfuscated Code

Yiannis Pavlosoglou
Brief Outline

1. Background
2. Motivation
3. Methodology
4. Fingerprinting Tool
5. Examples
6. Conclusions
1. Background

- Java Bytecode Operations
- Language Security Mechanisms
- Disassembling HelloWorld.java
- From Bytecode to Source
- What Popular Obfuscators Offer?
1.1 Java Bytecode
1.2 Language Security Mechanisms

- Type, memory and control flow checks
  - Instruction sets of the virtual machine [1]
    - Object creation
    - Privilege escalation [2]
  - Function calls
  - Exception handling
  - Verification (format, type, other violations) [3]
1.3 HelloWorld in Bytecode

HelloWorld.java

HelloWorld.class

HelloWorld.bc

(javac)

HelloWorld.java

SourceFile@HelloWorld.java

Hello World

Code:
0: aload_0
1: invokespecial #1; //Method java/lang/Object.<init>
4: return

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

Code:
0: getstatic #2; //Field java/lang/System.out
3: ldc #3; //String Hello World
5: invokevirtual #4; //Method java/io/PrintStream
8: return
1.4 Bytecode to Source

• Java is Platform Independent Code
  – Simplifies reversing compared to C/C++

• Popular Java Decompilers [5]

<table>
<thead>
<tr>
<th>Tool</th>
<th>License</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>DJ Decompiler</td>
<td>Free to try</td>
<td>2007</td>
</tr>
<tr>
<td>JAD</td>
<td>Free for noncommercial use</td>
<td>2006</td>
</tr>
<tr>
<td>JODE</td>
<td>GNU Public License</td>
<td>2004</td>
</tr>
<tr>
<td>Mocha</td>
<td>Free</td>
<td>2001</td>
</tr>
</tbody>
</table>

• Particular obfuscators are written with the operations of java Decompilers in mind [6]
1.5 What do obfuscators offer?

- **Operations:**
  - Less Human Readable Code
  - Remove Debug Information
  - Alter the Control Flow
  - “Encrypt” Constants
  - Restructure Simple Logic
  - Inject Unnecessary Code

- **Usability:**
  - Used for applications that are delivered to the user
  - Generally, used in J2SE Applets, Installed Applications, etc.
  - Generally, not used in Java Enterprise Environments
2. Motivation

- Basic Obfuscation Techniques
- Reverse Engineering Particulars
- Who is Using Obfuscators?
2.1 Basic Obfuscation Techniques

“Obfuscation is possible for the same reasons that decompiling is possible” [7]

- Renaming of objects and methods
  - a.class, b.class, a.c(5), etc.
- Extending classes
  - public class a extends d {
- Removing line numbers
- Encoding String Values
- Variable Splitting Techniques
  - boolean z becomes int x=1; if (x < 0)
2.2 Reverse Engineering Particulars

- Obtain Application Code
- Extract/Review Class Files
- Quickly Establish:
  - Renamed files
  - Missing debug info
  - Unnecessary code present
  - Potential code optimization
2.3 Who is Using Obfuscators?

- Let us have a quick browse...
3. Methodology

- A Taxonomy of obfuscating transforms
- String and constant representation
- Uniformity across the code base
- Kirchoff’s principle reversed
- Obfuscation encryption levels
# 3.1 Taxonomy of Obfuscating Transforms

<table>
<thead>
<tr>
<th>Data Obfuscation</th>
<th>Layout Obfuscation</th>
<th>Control Obfuscation</th>
<th>Preventive Obfuscation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split Variables</td>
<td>Scramble Identifiers</td>
<td>Clone Methods</td>
<td>Exploit Weaknesses in Current</td>
</tr>
<tr>
<td>Promote Scalars to Objects</td>
<td>Change formatting</td>
<td>Reorder Statements and Loops</td>
<td>Decompilers and Deobfuscators</td>
</tr>
<tr>
<td>Inheritance Relations</td>
<td>Remove Comments, Line Numbers</td>
<td>Reducible to Non-reducible Flow Graphs</td>
<td>Explore inherent problems with known</td>
</tr>
<tr>
<td>Change Encoding</td>
<td>Remove Debug Information</td>
<td>Extend Loop conditions</td>
<td>deobfuscation techniques</td>
</tr>
<tr>
<td>Split, Fold, Merge Arrays</td>
<td></td>
<td>Outline Statements</td>
<td></td>
</tr>
<tr>
<td>Reorder Instance Variables</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [8]
3.2 String and Constant Representation

• The Perfect Entry Point!

• Identifying Strings while Reversing:
  – Yields Architecture
  – Yields Design Patterns
  – Hard-coded Constants

```java
private static final String password = "ThisIsMyPassword";
...
if(args[0].equals(password))
    System.out.
    println("Password Correct!");
...
if(args[0].equals(z[2]))
    System.out.println(z[1]);
...
String as[] = new String[4];
as;
    as[0] = "qDX\027NSS\013%SSXY";
```
3.3 Uniformity Across the Code Base

- Typically, obfuscation is uniform across the code base
- Understanding how UI and IO operations are obfuscated is key to cracking
- Design patterns and used structures (e.g. Vectors, Lists, FILOs) are also key
- Generally, obfuscators do not offer more than a handful of options for a transform
3.4 Kirchoff’s Principle Reversed

• Obfuscation aims to make the code harder to interpret
• The equivalent of a known plaintext attack in a cryptosystem
• Becomes a known code attack for an obfuscator
• Using the obfuscator as a black box, assess the level of leet it offers
3.5 Obfuscation Encryption Levels

• Non-existent
  – Difficult to implement
  – Of little benefit: The bytecode has to run!

• No public/private crypto offered
  – Can it be implemented?

• String encryption uses XOR type operations
  – CPU and memory usage factors
4. Fingerprinting Tool

- Calibration Check
- Developing elucidate
- List of Available Flags
- Target Deliverables of elucidate
4.1 Calibration Check

• Depending on the obfuscation transform we intend to identify
• Attempt to generate generic definitions within a test class
• Objective: Capture the obfuscator’s fingerprint

```java
package elucidate;

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

    private static final String check =
        "\030\001\002\003\004\005\006\007\008\009\010\011\012" +
        "\013\014\015\016\017\018\019\020\021\022\023\024\025" +
        "\026\027\030\031\032\033\034\035\036\037\038\039\040\041" +
        "\042\043\044\045\046\047\048\049\050\051\052\053\054\055" +
        "\056\057\058\059\060\061\062\063\064\065\066\067\068\069" +
        "\070\071\072\073\074\075\076\077\078\079\080\081\082\083" +
        "\084\085\086\087\088\089\090\091\092\093\094\095\096\097" +
        "\098\099\100\101\102\103\104\105\106\107\108\109\110\111" +
        "\112\113\114\115\116\117\118\119\120\121\122\123\124\125";

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

- Build Calibration Classes for Particular Obfuscation Transforms

- Obtain Fingerprints for Particular Obfuscators

- Check the Generality and Overlap with other Obfuscators of that Fingerprint

- Include Results in elucidate
4.3 List of available flags

- `>perl elucidate.pl -h`

Elucidate v0.1 - Java Obfuscator Fingerprinting/Cracking Tool
Usage: elucidate.pl [-options *]

- `-h` : Print this usage message
- `-v` : Verbose option
- `-o` : Print supported obfuscators
- `-t` : Test current java environment
- `-f file` : Specify class file to identify
- `-j jar file` : Specify jar file to identify
- `-d directory` : Specify directory to identify

Examples:

- elucidate.pl -f MyClass.class
- elucidate.pl -d MyJar.jar
4.4 Target Deliverables of elucidate

- Given a jar file, or class files
- Identify which obfuscator has been used
- Recover known Strings within the file
- Give an estimate of the complexity
- Provide a map, as a tool of the application
5. Examples

- Examine the following commercial tools:
  - Zelix KlassMaster (4.5.0)
  - JShrink (2.3.7)
  - RetroGuard (2.2.0)
5.1 Zelix KlassMaster 4.5.0 (1/4)

- String literals three levels: Normal, Aggressive and Flow Obfuscate.
- PasswordCheck.class through javap:

```
6 :  ldc  #8; //String ,bw:) q`i  qv,=  "
14 :  ldc  #6; //String (km:N}?ps&,
22 :  ldc  #3; //String ,oa(-↓#w<.♀o}i?
30 :  ldc  #9; //String ,bw:)!!q`i←♬qk;
```
5.1 Zelix KlassMaster 4.5.0 (2/4)

- Output in Unicode format
- Special characters such as \n \b ...
- Unicode octal (\777)
- Unicode hexadecimal (\FFFF)
5.1 Zelix KlassMaster 4.5.0 (3/4)

- Uses XOR operation with five keys:
  - 124, 3, 4, 73, 94
- Original:
  ,bw:)!!q`i↔!!qv,"
- Decoded: Password Correct!

```java
64:  tableswitch{ //0 to 3
    0: 96;
    1: 101;
    2: 105;
    3: 109;
    default: 114 }
96:bipush 124
98:goto 116
101:  icall
102:  goto 116
105:  icall
106:  goto 116
109:  bipush 73
111:  goto 116
114:  bipush 94
116:  ixor
117:  i2c
```
5.1 Zelix KlassMaster 4.5.0 (4/4)

- 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.
5.2 JShrink 2.3.7 (1/4)

- Creates a new package, with a single class
- Replaces String code with:
  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
- Equivalent to: ( I.I.I( 79 ) );
5.2 JShrink 2.3.7 (2/4)

- Contents of package I include a file called I.gif
- However, this file is not an image file:

![File contents](image)

- As a file is being accessed, a decompiler can be used to view I.class
public class I {

    ... 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' );
        }
    }
}
5.2 JShrink 2.3.7 (4/4)

- Creates an invalid gif file storing the encrypted Strings
- Uses a separate class and method to perform decryption
- Replaces Strings with: I.I(int) e.g. I.I(97)
- Does not alter Strings declared as static and final
- Introduces exceptions if the wrong int is passed as argument
5.3 RetroGuard 2.2.0

- Does offer String encryption
- Goes to show that some obfuscators simply don’t use this approach
- The creators of RetroGuard quote: “obfuscation is not encryption”
6. Conclusions

- Static obfuscation is at a primitive level
- Encrypted Strings are an excellent entry point into understanding the application
- Identifying the crypto used:
  - yields the obfuscator tool used
  - yields what changes to expect in snippets
6. **Final Conclusions**

- Propose polymorphic obfuscation
  - Developers map out critical elements
    - Understanding of what an obfuscator can do
  - Obfuscator changes behaviour depending on file
    - UI treated differently to say, protocol implementation
  - Algorithms vary according to key file
    - In how many ways can you write a for/while loop?
Questions

• subere@uncon.org