Methods for Obfuscating Java Programs

Florin BUZATU
IT&C Security Master
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies
ROMANIA
florinbuzatu2005@yahoo.com

Abstract: Java programs distributed through internet are suffering of problems like reverse engineering, algorithms theft and unauthorized functional changes. The reason for this issue is that the bytecode from Java class file holds enough information to be decompiled into source code that resembles the original. In this paper, I discuss some practical obfuscation methods that make difficult the understanding of the decompiled programs. The methods analyzed are: lexical obfuscation, data obfuscation, control flow obfuscation, obfuscation methods that make use of the discrepancy between bytecode and source code. In order to evaluate the usefulness of these methods, some metrics will be described.

Key-Words: Java, obfuscation, lexical obfuscation, data obfuscation, control flow obfuscation, bytecode

1. Introduction

Most of applications created today are targeted to virtual machines such as Java virtual machine or Microsoft Common Language Runtime. These platforms offer a number of benefits such as portability, standardized libraries and automatic memory management. But these benefits also come with a cost: as opposed to languages such as C or C++ that are compiled in machine code that is difficult to reverse engine, a programming language as Java targeting a virtual machine is compiled to an intermediate representation called bytecode [3]. In Java, dependencies are resolved at runtime when classes are loaded by the virtual machine. This implies that field, methods and class names must be kept in the bytecode. Using a decompiler we can obtain the source code resembling or even identical with the original. The bytecode can be easily reversed engineered, so a hacker can have access to ideas, algorithms, data structures, security and licensing mechanism or can even alter the purpose of the program [7].

There have been designed various ways to ensure the intellectual property of Java programs. Some of them, like using an AOT (ahead of time) compiler to transform the bytecode [6] into a native executable defeat the purpose of using a virtual platform: portability. Others, like encrypting the bytecode are fundamentally flawed as there are well known points where the unencrypted bytes can be intercepted. The protection can be skipped without even to try to break the encryption. One other way to secure a program is through obfuscation. This method has also some disadvantages, like some of the obfuscation methods can slow down a program, but it also has advantages such as: renaming variables to shorter names decreases the size of the program, which leads to a shorter time to load the bytecode.

An obfuscator is a program that applies transformations on the bytecode (in some cases on the source code) in order to make it difficult to analyze an application and to decompile it. Obfuscation tries to conceal the purpose of an application while keeping intact its functionalities.

Functionalities of an obfuscator are:
- (1) lexical obfuscation – class, field and method names are replaced with names without meaning;
- (2) data obfuscation – makes difficult the understanding of data structures. It alters the way variables are stored.
in memory and the way they are interpreted;

- (3) control flow obfuscation – modifies the control flow of a program. Obfuscation can be achieved also by exploiting the discrepancies between what can be expressed at bytecode level and what can be express at source code level;

- (4) watermarking the program [9] – watermarks can be inserted in a program with the purpose of identifying the owner of the application. Watermarks can be used in multiple ways: if in the build process a watermark is added to identify the customer that purchased the application, and pirate copies appear on internet then the customer disobeying the license can be identified. Otherwise by inserting the producer, one could prove if competition tried to steal the algorithms;

- (5) string encryption or encoding – although not a full proof method, by encrypting strings can make the opponent work more difficult. On the other side this is not an irreversible method;

- (6) application shrinking – unused classes, methods or fields can be removed. If the application uses third party libraries the number of classes can be reduced considerably. Also from the bytecode can be removed unused annotations or generics information;

- (7) debug info obfuscation – line numbers can be removed or mangled;

- (8) library support for APIs like RMI, JavaBeans, EJB or serialization;

- (9) stack trace translation – stack traces from applications used in production are mostly useless unless there is a way to locate error source;

- (10) Ant tasks for integrating the obfuscation in the build process.

2. Problem definition

An opponent can interact with the application in the following ways:

- He can study the program to extract knowledge;
- He can try to reuse methods and algorithms;
- He can change the behavior of the application by making changes that do not obey the licensing terms;

An obfuscator attempts to make the source code generated by reverse engineering harder to understand and less useful for hackers and competition.

3. Problem solution

3.1. Lexical obfuscation

Lexical obfuscation or names obfuscation is the process in which the identifiers of classes [1], fields and methods from the bytecode are replaced with new meaningless identifiers. As such, a class named for example

com.mycompany.installer.LicenseController

can be renamed to

a.a.a.A.

To ensure consistency all occurrences of a name must be replaced with the new name. Java programming language supports method overloading (two or more methods can have the same name if the argument lists are different). This can be speculated by renaming unrelated methods to the same name. For example

getBook(String name)
setAddressor(String book, String author)

can become

a(String a)
a(String a, String b).

Obfuscators must be designed so they can generate names correctly even in the case of complicated inheritance hierarchies and when multiple interfaces are implemented.

Name obfuscators can generate shorter names with the goal to reduce the size of the application. This reduces the time to load applications from internet and the time to start desktop applications.

Name obfuscation also has some disadvantages:

- Names of standard java classes cannot be obfuscated (java.* and javax.* packages);
- Classes used through reflection or that use native methods cannot be renamed. Some frameworks like JavaBeans or EJB are based on reflection;
- Some fields and methods from serializable class names cannot be obfuscated.

3.2. Data obfuscation

Data obfuscation modifies the data structures used by the application. There are multiple ways to achieve this:

- By the way data is kept in memory. For example a local variable can become a global variable [5],
- Constant variables can be embedded as fields. Strings are frequently used as constants throughout the code. By moving the string literals into fields, the code can convey less meaning. To prevent the undoing of this obfuscation, the initialization of the field can be further obfuscated by using an opaque predicate (expression that evaluates to true or false)
- Local variables can be packaged into bit fields. This method can be used to obfuscate local variables of primitive types. Two or more primitives can be packed together into a variable that has more bits. Read or write operation must be replaced with packaging and unpackaging operations, so this would slow down the application.

- The way data is grouped can be changed. For example a matrix can be transformed into an array or vice-versa.
- The way data is ordered can be changed. Data from an array can be stored at a different index. A function can be used to calculate the new index.
- Arithmetic expressions can be converted to bit shifting operations, which also might improve performance.

Although a decompiler is expected to produce compilable Java source code for these obfuscations, this might not be the case for all, as some of the above obfuscations methods might exploit a semantic gap between: at bytecode level booleans, bytes and chars are expressed as integers.

3.3. Control flow obfuscation

Obfuscating the control flow means modifying an application so it produces the same results at runtime, but is harder to decompile it in well structured source code. Obfuscating the control flow makes transformations such as separating calculations logically belonging together, grouping unrelated calculations or inserting irrelevant code.

One notion used when presenting this kind of obfuscations is the opaque predicate notion. An opaque predicate is an expression that evaluates either to true or false – the outcome is known a priori, but is still evaluated at runtime.

Control flow obfuscation methods are:

- Inserting unused or irrelevant code. The complexity of a piece of code is directly proportional with the number of predicates it uses. It is possible to insert a predicate that is always true and so to split in two a code block. One branch executes the original code, the other some dummy code. The inserted predicate increases the code complexity, but it does not modifies program output. In the same idea, a block of code can be replaced by a predicate with two branches. The predicate can evaluate
to either true or false and the two blocks produce the same results without being easily noticeable.

- Code parallelization increases code complexity. This method is highly resilient and potent as it is difficult to understand the code for the programmer and for the decompiler.
- Method inlining – method calls are replaced by the method code. This is usually used as an optimization technique as the call overhead is eliminated. Method outlining is the reverse process in which some instructions are packed into a method. These solutions can be combined by replacing two methods with their code and then creating a new method that contains the last instructions from the first methods and the first instructions from the last method.

3.4. Exploiting the design gap

There is a certain discrepancy between Java language and what can be represented at bytecode level. The classical example is the goto instruction that has no equivalent in the source code. This discrepancy can be speculated by obfuscators to transform the code in such a way that decompilers cannot produce correct source code. Obfuscating methods in this category are [1]:

- On a branch of an if instruction a jsr instruction is added. The jsr instruction calls a subroutine defined in the body of a method. This instruction adds on the stack the address of the instruction immediately after it, then it jumps at the address indicated through a parameter as an offset. The address on the stack is usually used as by the ret instruction. For obfuscation at the target indicated by the if instruction two instructions are added: jsr and pop.
- Reordering load instructions before if instructions: this obfuscation can be used when on both branches of the if instruction the same local variable is loaded onto the stack. This is a pretty common occurrence, if we think at an example as if (condition) then i = value1 else i = value2. The obfuscation moves the load instruction above the if.
- Ignoring constructor conventions: Java language specifications demand that the first instruction from the constructor to be a call to another constructor of the same class or to the parent class constructor. If this instruction is missing is automatically inserted by the compiler. This restriction does not exist at bytecode level, which leaves opened the possibility to insert instructions so a decompiler will not manage to translate the bytecode in valid source code. There are multiple ways to create code that will fool the decompiler: insert before the super call jsr and pop instructions; wrapping the supper call in a try block; in classes that extend java.lang.Throwable insert a throw instruction before the super call. This throw instruction is wrapped in a try block that has as handler block a super call. This method speculates that exceptions can be added on the stack using a throw instruction, not just a standard load instruction.
- Guarding a part of a switch block with an exception handler. There are no bytecode instructions that catch exceptions. The bytecode of a method is associated with a list of exception handlers for exceptions that specify what code must be executed when an exception is thrown in a certain part of the method. At bytecode level a try catch block is represented by a domain with a list of instructions belonging to the try block and a handler that is the equivalent of the catch block. The domain is specified by start and end labels, and the handler by a start label. If an exception is thrown in the try block, the stack is cleared and then the exception is pushed onto the empty stack and the execution continues at the catch block. A switch instruction pops from the stack a value and then jumps at
the address indicated in the jump table. It is possible to guard with an exception handler only a part of the code generated for a switch, which makes it impossible for the decompiler to generate correct source code.

- Combining try blocks with catch blocks: in the source code the try catch blocks can be represented in a single way – with a try block followed by one or more catch blocks. At the bytecode level, the try blocks can guard the same bytecode used to handle the exception. The try and catch blocks can be combined so they have the same beginning. An if instruction is inserted that a branch has the try code and on the other the code from the catch block. Once first branch is entered, a flag is set to indicate that in the future the execution must follow the other branch. This flag is reset at the end of the catch block.

- Adding goto instruction: the goto instruction is used at bytecode level to implement for and while cycles. The goto instruction is not allowed at source code level. Adding a goto instruction at the target of an if instruction can be an attempt to obfuscate the bytecode. Still, a decompiler that uses control flow analysis can produce valid source code. To obfuscate the code in an efficient manner, a solution is the following: A method is split in two parts – A1 and A2. The generated code is goto A1; A2; A1; goto A2. To complicate things even further a try block can be added from the end of A2 to the beginning of A1.

3.5. Metrics for evaluating obfuscation methods

To study the obfuscation methods we should have the possibility to evaluate the quality of the transformations. Some of the metrics that can be used to evaluate the are [8]:

- **The potency**: this is the degree in which a transformation misleads a human engineer that tries to understand the program. For example, in the case of source code obfuscation, a high degree of potency is archived by removing the formatting.

- **The resilience**: it defines how hard the transformed code can be deobfuscated by using tools that automate the operation. The highest degree of resilience is archived by one way transformations that are not reversible, such as changing names or removing from bytecode the line numbers of the corresponding source code.

- **Stealth**: defines how well an obfuscated code matches with the rest of the code. If the transformation produces code that comes out easily, the code cannot be identified easily by an automated tool, but it can be spotted by an engineer.

- **Cost**: represents the overload added to the execution time compared to the unobfuscated application. In some situations, the obfuscation improves the execution time. For example, renaming identifiers to shorter names reduces the time to load the classes in memory and improves application startup. On the other side, there are obfuscations that slow down the program.

4. Conclusion

In this paper are presented four groups of obfuscation methods that operate on the Java bytecode level. The transformations performed by an obfuscator raise the reverse engineering bar, making the programs harder to understand, analyze and modify.

References


[2] Eric Bruneton, ASM 3.0 A Java bytecode engineering library


