International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 06 Issue: 04 | Apr 2019 www.irjet.net p-ISSN: 2395-0072

OBFUSCATION: MAZE OF CODE

Brijesh Patel[1], Vishal Patil[2], Karan Lohar[3], Yogita Mane[4]

[1,2,3]*Student, Dept. Of I.T Engineering, Universal College of Engineering, Maharashtra, India*
[4]*Professor, Dept. Of I.T Engineering, Universal College of Engineering, Maharashtra, India*

---***---

Abstract - *Software Reverse Engineering scenario would involve software that has been worked upon for years and carries several modules of a business in its lines of code. Unfortunately, the source code of the application or software cannot be recovered easily; what remains is "native" or "binary" code. Traditional obfuscators work on binary code, but they are tedious and do not provide us with a specific obfuscation depending upon the code. We provide a way in which the code can be obfuscated depending upon lines of code and variables. We propose the application of this data confers additional information developers need for better understanding or knowing, maintaining and developing software in large team settings. This solution is proposed in order to discourage software piracy and impede malware analysis. This book introduces Code-a- Maze, reveals the results of our evaluation and proposes directions for future research in this area. Code-a-Maze will increase size of code and compilation time is also satisfactory.*

***Key Words*: obfuscation, reverse engineering, de-compilers, allocation and deallocation, dummy calls, bogus code, trampolines, byte code.**

1. INTRODUCTION

Reverse engineering commonly called as back engineering is a way of obtaining knowledge regarding a code or an algorithm (in case of a particular computer program). The reasons to obtain such information may either be good or bad depending upon a particular situation. People belie saying reverse engineering process in itself is not concerned with creating a copy or changing the artifact in some way; it is only an analysis in order to deduce design features from products with little or no additional knowledge about the procedures involved in their original production. There are three types of Reverse Engineering software, hardware and producing 3-D images. In this book, our focal point is on software reverse engineering.

The actual purpose of software reverse engineering is done to retrieve the source code of a program because of various reasons which are the source code was lost, to study how the program performs certain operations or tasks, to enhance the performance of a program, to fix a issues or bugs (correct an error in the program when the source code is not available), to identify malicious content in a program such as malware or virus. Summarizing, the main aspect of software reverse engineering is to see how code functions to enhance or protect it from harm. To retrieve source code (java file) when lost or to enhance the functionality we use de-compilers to convert the .class file to java code and make improvements. The normal Java compilers considering JVM, does not particularly obfuscate the code while making a .class file. Hence, it becomes easy for the intruder to add malicious content or copy the source code by using de-compilers. Although clearly stated Reverse engineering for the purpose of copying or duplicating programs may constitute a copyright violation, but people still back track the code for various social benefits.

In order to prevent these various obfuscators where created to ensure security of various software's. These obfuscators generally confuse the intruder about the main functionality of the code and do not provide the initial source code which is optimal. The obfuscators increase a lot of overhead, but they are still used to protect the integrity of the code. However, these obfuscators change the code flow and some of those changes make it impossible for the JVM to efficiently optimize the code. In effect it will actually degrade the performance of your application. And hence our approach is to work on the efficiency by providing a MAZE in which only the required obfuscation occurs depending on various factors of code.

Intermediate-level obfuscation, which obfuscates a program at intermediate representation (IR) level, which is typically used for interpretation-based language such as Java is the one which we are trying to focus on. This book covers the software reverse engineering problem scope, our approach to raising awareness about various faults in the exciting systems and finally the approach to tackle Software Reverse Engineering in java codes in a simple way.

1.1 Aim and Scope

Reverse Engineering explains the process or ways of determining the inner workings or set of rules followed by an engineered piece of kit (hardware or software) in the absence of design plans.

Sometimes, reverse engineering is essential or unavoidable. For example, when the working or data about a certain product has been lost, reverse engineering could be done so as to recover the lost data so that product maintenance may continue and proceed further. Another example of reverse engineering is used in spying, espionage and scientific

research related to warfare that is military purpose; countries or parties involved in a war (or potentially involved in a war) typically wish to understand or gain information about the inner workings of the enemy's weapons or communication methods to improve defenses or prepare efficient means of attack.

Today, however, reverse engineering is most commonly associated or understood as the cause of theft of intellectual property, hacking. Someone might purchase an engineered kit (Software or Hardware) from the original manufacturer, take it apart and analyzing it or understanding its working to re-build clones or copies of the original device without investing into development and research. Hence Reverse engineering has its own pros and cons and it could be both beneficial or hazardous as it could have quiet an effect on a company's growth as well as its downfall, hence this cons must have some countermeasures and one of it is in form of code obfuscation.

1.2 Software Reverse Engineering (SRE)

Having many of its advantages can be exploited for injecting malicious code, viruses, malware e.g. Code Red worm. Now to mitigate security problems and software privacy saved by SRE, we will devise a technique to deter control flow by making information flow obscure. This is achieved through flipping conditional branches, obfuscating control transfer and inserting bogus code. The code injection will be done using branched functions, data pointers and structures, pointer allocation- deallocation, dummy calls, trampolines and start-up routines. This can be achieved at intermediate level. To achieve the above goal, we intend to design a UNIVERSAL Code-a-Maze which will be suited for multiple languages and dynamically generated. The maze will have multiple entry-points and the entry to the code will be transferred to a different point after a specified time. The design of the maze will have modest compilation time and maximum efficiency. We will do a detailed analysis of performance of our system.

2. Existing System

The one feature that distinguishes Java from other high-level programming languages is its platform independence. This characteristic is possible because of Java Virtual Machine (JVM). The Java compiler coverts the human-readable java code to an intermediary byte code. This byte-code is then read by JVM to generate machine specific codes. The intermediary byte-code, generated by the javac compiler, is stored in a .class file. However, if any attacker gets hold of the .class file, he/she can use ready-made de-compilers and directly get access to the source code, hence compromising security. This is where obfuscators come into the picture. Following are the commonly used obfuscators available in the market:

2.1 ProGuard [8]

ProGuard is one of the few open source obfuscators which is also integrated in Android SDK. Along with obfuscation it also provides other options like shrinking, optimizing and pre-verifying your Java class file. It is basically a java obfuscator and can also be used for Android applications. ProGuard includes identifier obfuscation for packages, classes, methods, and fields. Without proper naming of classes and methods it is much harder to reverse engineer an application, because in most cases the identifier enables an attacker to directly guess the purpose of the particular part. It is being used by developers at companies and organizations like IBM, HP, Siemens, Nokia, Google, Intel, and NATO. Although it is used so widely, ProGuard can only be described as a modest deterrent. The obfuscation techniques used by ProGuard are not enough to stop attackers, but enough just to give them another hurdle to cross. The program code itself will not be changed heavily, so the obfuscation by this tool is very limited. Hence the overall level of security provided by ProGuard is deficient.

2.2 Allatori [6]

Allatori is a commercial obfuscator from Smartdec. Besides the same obfuscation techniques like ProGuard, it also provides methods to modify the program code. In order to thwart reverse engineering, loop constructs are segregated and merged, thereby achieving obfuscation. The less readable code and incremented length further augment complexity. Additionally, strings are obfuscated and decoded at runtime. This includes messages and names that are normally human readable and would give good suggestions to attackers. The obfuscation methods used in Allatori are a superset of ProGuard's so it is more powerful but does not prevent an attacker from disassembling an application. However, with increase in complexity of the code, the cost of the application also increases considerably. Also, its potency score mediocre compared to other obfuscators after code transformation.

2.3 GuardIT [7]

GuardIT is a proprietary obfuscator by Arxan which not only performs code obfuscation, name obfuscation and string encryption but also goes above and beyond to wrap the application code into an active protection system capable of detecting tampering in real-time. These advanced techniques operate on bytecode thus making it for Java non-interoperable with AOT compilers. Sophisticated code obfuscation transformations move, split, insert, and create functionally equivalent code sequences which thwart de-compilation and reverse engineering of the underlying byte code itself.

3. Proposed System

Our approach for obfuscation is not to alter the class file (byte code) but to provide an obfuscated code before compilation. We are mainly focusing on Java code using JVM for compilation. In the below figure 4.1 we show how our obfuscation method would be suitable. In this approach:

1. First, we generate or retrieve a java file considered as source code which we pass it through MAZE obfuscator.
2. The output of the obfuscator is an obfuscated code which is our new source file and it can be in .txt format, .java format or any other format which can be later converted to .java file.
3. The obfuscated code passes through JAVA VIRTUAL MACHINE (JVM) along with the previous source code. On compiling, we receive .class files of both the source code and obfuscated code as the output.

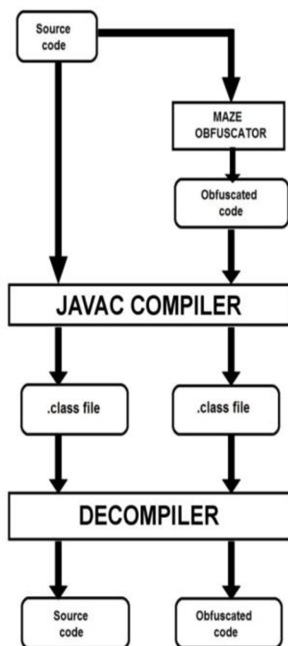

Fig -1: Flowchart Obfuscator

4. When the intruder or a person who wants to retrieve the functionality of code and manages to get access over the compiled file, he can use a De-compiler.
5. The De-compiler provides the source .java file which helps the intruder to know the functionality and working of code.
6. However, in our case the intruder receives the obfuscated code which is also difficult to understand and traps the user eventually.
7. In this way we can stymie understanding of our code even if the intruder has access to .class file.

Further, we are explaining the functioning of MAZE obfuscator.

3.1 System Architecture

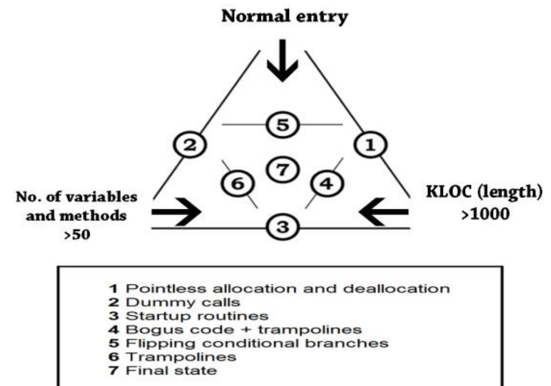

Fig -2: Maze of Code

In order to hinder analysis of the source code and enhance obfuscation, we have designed a Code-a-Maze that can be integrated with the source code available and thus the complete structure is metamorphosed. The working of the MAZE is as follows:

1. First, we evaluate the given source code based on parameters like thousand lines of code (KLOC), number of methods and variables, etc. If it surpasses the minimum threshold, then it enters the MAZE depending upon the condition satisfied and proceeds further in the MAZE.

2. Next it is subjected to various transformations explained below, which modifies the code depending upon the entry of the path analogous to a maze.

3. After the application of the proposed variations and modifications, the flow of the code proceeds further, ultimately reaching the final state or goal state.

Thus, the aim of the MAZE is to complicate the code, thereby preventing its analysis without obstructing the natural flow of the source code. The various transformations or modifications for different paths in the MAZE are as follows:

1. **Path 1:** In this path, the code is subjected to *pointless allocation and deallocation*. In this method, the memory allocated for various classes and variables in the code is changed. This is done dynamically where the stack changes using copy constructors, assignment operators and destructors. (Refer to the below reference).

2. **Path 2:** In this path, the code is subjected to *dummy calls*. Dummy calling involves unnecessary use of class definitions in a method. It simply calls functions and methods defined in the main class,

whose function is analogous to a round-about, wherein no new changes are made, just shifts the control to a temporary state and returns back to normal state.

```
public class TestClass
{
    public static void main(String[] args)
    {
        TestClass t = new TestClass();
    }

    private static void testMethod(){
        abstract class TestMethod{
            int a;
            int b;
            int c;

            abstract void implementMe();
        }

        class DummyClass extends TestMethod{
            void implementMe(){

            }
        }

        DummyClass dummy = new DummyClass();
    }
}
```

Fig -3: Dummy calls

3. **Path 3:** Through this path, the code is subjected to *startup routines*. We can write our own startup routine codes, which makes it difficult for debuggers to bind. The sole purpose of startup routines is to transfer control flow form one function to another, making it difficult for any intruder to understand flow of data.

```
void startup();
int _start()
{
    startup( );
    exit   (0)
}
void startup()
{
    /* code here */
}
```

Fig – 4: Startup routines

4. **Path 4:** This path is a combination of two techniques namely, *bogus code* and *trampoline.* Some specious lines of code are added having no significance, wherein the condition is always satisfied. It is intentional distortion of code through fabrication of program statements. Secondly, the trampoline function makes the control jump to different temporary state and returns.

```
if  (x*x >= 0)

{s1;

}

else {

s2;//bogus code

}
```

5. **Path 5:** This path describes about the *flipping conditional branches*. In this approach certain pre-defined conditions are executed, and it results in shift of program control. The net effect is that the code executes certain extraneous conditions to get the same result as the natural flow of the code.

6. **Path 6:** As mentioned earlier, this path introduces a *trampoline*, where the function control is jumped to a higher state and brought back acting like a trampoline. Once the given condition is satisfied, the code shift takes place and ultimately is brought back to the same initial state with little or no modifications.

```
 void trampoline(void (*fnptr)
(), bool ping = false)
 {
   if(ping)
     fnptr();
   else
     trampoline(fnptr, true);
 }
```

Fig – 5: Trampolines

7. **Path 7**: It is the final or goal state of the MAZE. Every transformation ultimately desires to reach this state through various paths depending upon the conditions and parameters. Once the control is in this state, it output is same as the output of the source code.

Thus the designed Code-a-Maze achieves the aim of obfuscating the given source code, applying various transformations and modifications to hinder easy analysis of code and in turn prevent SRE.

3.2 Module wise algorithm / Pseudo code

I) Switch on NetBeans and Run Java program
II) Accept the Input File.
III) Check for the 2 conditions<lines of code, number of methods>.
III) Retrieve all function names from the code.
IV) Generate random string for each function name while forming Key-Value Pair.
V) Replace all function names and function calls with corresponding random strings.
VI) If numbers of functions are more than 50:

A. Select one from dummy calls and start-up routines at random and make the function call.
B. Insert the string returned from the function at appropriate position in the code.
C. Call the trampoline function.
D. Insert the string returned from the function at appropriate position in the code.

VII) If lines of code are more than 1000:

A. Select one from pointless alloc-dealloc and start-up routines at random and make the function call.
B. Insert the string returned from the function at appropriate position in the code.
C. Call the bogus code function.
D. Insert the string returned from the function at appropriate position in the code.

VIII) If numbers of functions are less than 50 and lines of code are less than 1000:

A. Select one from pointless alloc-dealloc and dummy calls at random and make the function call.
B. Insert the string returned from the function at appropriate position in the code.
C. Call the trampoline function.
D. Insert the string returned from the function at appropriate position in the code.

4. Benefits of the Proposed System

1. *Platform Independence*- the transformations performed on the code are independent of the nature of application and are applied on high-level code.

2. *Diversity*- because of wide range of techniques available for obfuscation, different instances of the same original code can be created thereby making it difficult for attackers to intrude.

3. *Protection*- obfuscation provides protection against static and dynamic attacks, in turn raising the bar for the attacker, as the attacker requires extra time and resources to crack the obfuscated code.

4. *Low cost*- with the automation of various transformations and compatibility with the existing systems, the obfuscated code involves low maintenance cost and efficient use of resources.

5. *Simple*- the obfuscation techniques are comparatively easy to apply and achieve security of confidential code.

4.1 Future Scope

- **Extending the idea to dynamically generate the maze**:
  The algorithm that we have developed can be further used to generate the maze dynamically based on the level of complexity required by the code.
- **Using the Maze to obfuscate any kind of application/code**:
  With our proposed approach we intend to convolute any kind of web and/or android application and even made it compatible across multiple platforms.
- **Developing software for Digital Rights Management (DRM):**
  Since the algorithm can be extended to implement access control by integrating various encryption techniques, our Maze can prevent piracy and be used in DRM. Thus, trying to control the usage, doing any modification, and distributing the copyrighted works (such as software and multimedia content), and also systems within devices that enforce such policies.
- **Extending the idea to be used in Intellectual Property (IP):**
  It is prominent that the protection of intellectual properties is a critical issue for the vendors while capturing customers with new technologies, with new software is the aim of every software vendors, but to protect their new ideas they need copyright or patent.

5. CONCLUSION

Our project successfully used the maze of code technique to implement obfuscation and hinder code analysis. Our proposed system processes any kind of java executable code within the constraints convoluting it with considerable increment in compile time. Thus, our maze reduces the computational time making it cost and time efficient for it to be implemented in wide range of sectors, by any organization.

REFERENCES

[1] Patrick Schulz – Code Protection in Android. University of Bonn, Germany. 2012

[2] Marius Popa - Techniques of Program Code Obfuscation for Secure Software. Journal of Mobile, Embedded and Distributed Systems, vol. III, no. 4, 2011.

[3] Matthew Karnik, Jefferey MacBride, Sean McGinnis, Ying Tang, Ravi Ramachandran – A Qualitative analysis of Java Obfuscation. Proceedings of the 10th IASTED International Conference Software Engineering and Applications, Dallas, Texas, USA, November 13-15, 2006.

[4] Kirti Mathur, Saroj Hiranwal - A Survey on Techniques in Detection and Analyzing Malware Executables - http://www.ijarscsse.com/docs/papers/Volume_3/4_April2013/V3I4-0288.pdf

[5] Jan Cappaert-Code Obfuscation Techniques for Software Protection - www.cosic.esat.kuleuven.be/publications/thesis-199.pdf

[6] Douglas Low – Java Control Flow Obfuscation http://www.cs.auckland.ac.nz/cthombor/Pubs/dlowthesis.pdf

[7] Arxan obfuscator system - https://www.arxan.com/resources/technology/app-code-obfuscation

[8] Proguard Obfuscator system - https://www.guardsquare.com/en/products/proguard

[9] Allatori Obfuscator system -