Code obfuscation for software security

Muhammadi Emin, fm5cc5@inf.elte.hu

1.0 Introduction

With the progressions in advanced innovations, the risk of the immeasurable degree of copying and unlawful repeating of applications is increasing. In this way, the number of security breaches is comparatively developing. Code obfuscation is an instrument used to obfuscate the local code, data structures, or native code. Overall, code obfuscation is about hiding the intricacies of a program's execution from an intruder, i.e., converting the untouched program into an identically (with the same computational implications) obfuscated one that is much unintelligible for an attacker to understand. Both analysts and software companies make a good-faith effort to update their intellectual property with fresher and superior obfuscation techniques on a customary cycle. However, programming code still tends to get broken. Obfuscation techniques are implemented at the assembly level, with the hypothesis of improving changes. Assembly code addresses the information conditions and solace to dissect the information in the wake of dismantling the executable when contrasted with the decompiled code.

In this paper, the importance of code obfuscation has been clarified, and during the experiment phase, the success of the obfuscation techniques was examined.

2.0 Identifying the problem

The main problem of software applications is the integrity of the binary code. If the binary code is modified, the software can be used to perform malicious activity. On the other hand, third-party users can also decompile the binary code to gain access to the source code and distribute it to other users illegally. The following scenario can happen in the real-life:

Software Development Company releases paid software. This software should be accessed only and only if a user has a valid license key. When the app starts, it should ask the user to enter a license key. If the user enters a valid license key, then the app should show the user's details and it starts to work. The license key is a string of characters. This license key should be verified by the software company via API endpoints. App sends a request to *example. localhost/verify* with license key as a query parameter. What if someone tries to hack the app without knowing the source code, and distributes a malicious version of the app on the Internet?

An intruder can easily analyze the binary software using one of the following open-source disassemblers and reverse engineering. After intercepting the application, it is possible to change some bytes of the code or even to remove some functions (insert malicious code into the program). Following steps are implemented using IDA[1] to reproduce the vulnerability:

i. Intruders can use a variety of techniques to steal information from a system. First, intruders need to analyze the system and find out what is the security weakness. Then, they can exploit the weakness and modify the binary code of the program.

ii. After learning some basics about software, intruders can search which strings are hidden in the code.

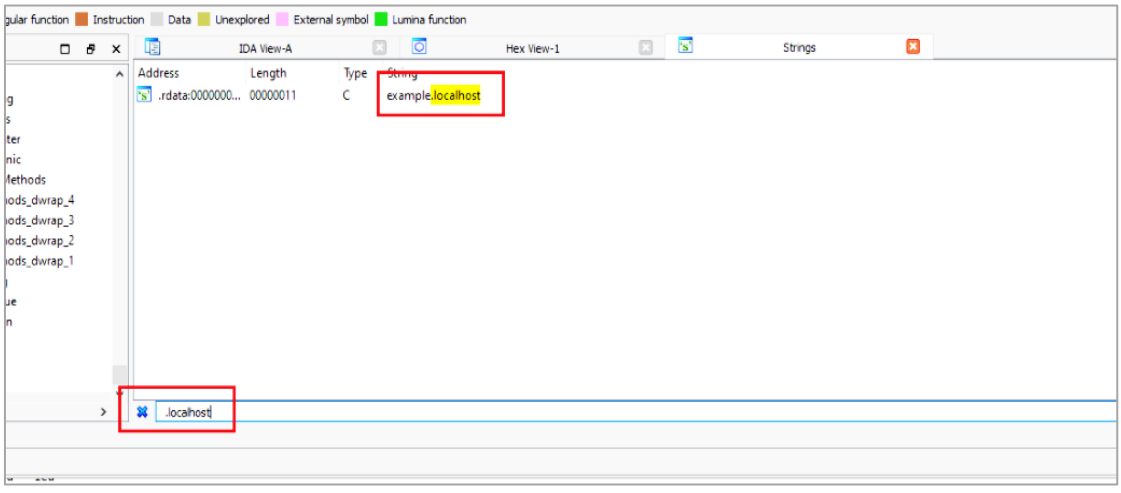

Figure 1 – String View on the IDA (Binary code analysis tool).

iii. Next, an intruder can change the bytes of code, which can be used to bypass some functions of the software. For example, in this case, the attacker can change the

[1] IDA, Binary Analysis Tool - https://hex-rays.com/ida-pro/

domain name of the website, which can be used to redirect or access all requests from the users of the modified software.

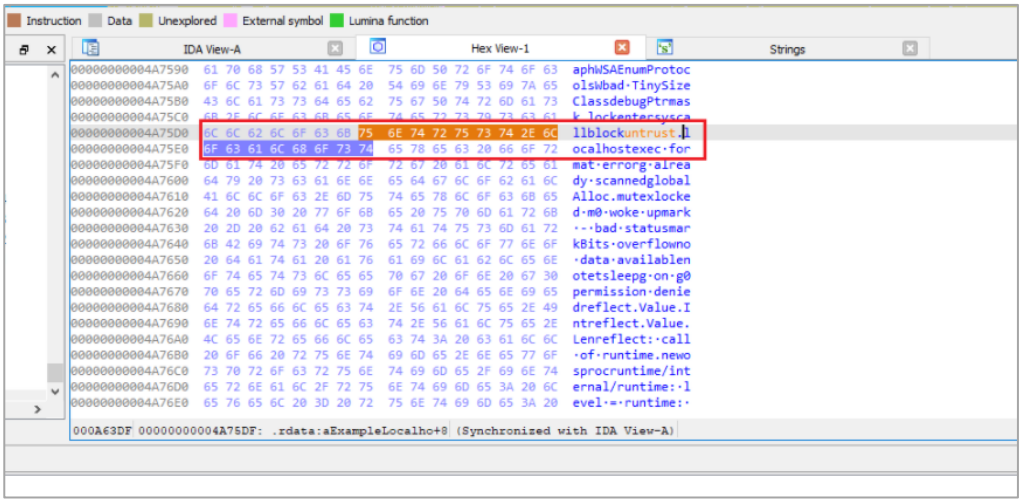

Figure 2 – Hex View on the IDA (Binary code analysis tool).

iv. Finally, the attacker applies the patch to the software and distributes it to the victims.

In most cases, the vulnerability can occur on the end node. It can be a cloud service or an end end-user device. Nobody is trustworthy when it comes to software security, thus engineers should ensure that software does not tamper with malicious code.

3.0 Obfuscation and its techniques

Obfuscation means to make something difficult to understand. Programming code is regularly obfuscated to secure intellectual property and keep an aggressor from reverse engineering a restrictive application. Encoding a few or all of a program's code is one obscurity technique. Different methodologies incorporate stripping out possibly uncovering metadata, replacing class and variable names with inane names, and adding unused or unimportant code to an application script. A tool called an obfuscator will consequently change over direct source code into a program that works the same way yet is harder to peruse and comprehend.

Code obfuscation uses complex indirect expressions and an excess of logic to make the code difficult for the intruder to understand. The goal is to sidetrack intruders with the confounded syntax of what they are perusing and make it difficult for them to choose the substance of the genuine code.

In a computer code, the intruder may be a person, a computing machine, or some other program. The confusion is also used to trick antivirus devices and other projects that rely heavily on advanced characters to decode code. Decompilers are available for languages like Java, working frameworks like Android and iOS, and development stages like .NET. They can consequently decode source code and make it hard for these projects to do their decompilation.

Obfuscation involves several different methods. Regularly, numerous methods are utilized to accomplish a layered impact. Programs written in compiled software languages such as C# and Java are easier to obfuscate. In view of the fact that they make mid-level guidelines that are commonly simpler to peruse. In contrast, C++ is harder to obfuscate because it is compiled into machine code that is harder for humans to handle. Common obfuscation techniques include the following:

- **Packing**. This compresses the entire program to make the code unreadable.
- **Renaming**. The obfuscator changes the methods and names of variables. The new names may contain unprintable or invisible characters.
- **Control flow**. The decompiled code looks like spaghetti logic, which is unstructured and hard to maintain code where the line of thought is obscured. The results of this code are unclear, and it is hard to see what the point of the code is when you look at it.
- **Instruction pattern transformation**. This approach exchanges common instructions produced by the compiler for more complex, less common instructions that effectively do the same thing.
- **Dummy code insertion**. Dummy code can be added to a program to make it harder to read and reverse engineer, but it does not affect the logic or outcome of the program.
- **Metadata or unused code removal**. Unused code and metadata give readers additional information about the program, much like annotations in a Word document, which can help them read and debug. Removing metadata and unused code gives the reader less information about the program and its code.
- **Opaque predicate insertion**. A predicate in code is a logical expression that is either true or false. Opaque predicates are conditional branches (if-else statements)

where the results cannot be readily determined with statistical analysis. Including an opaque predicate introduces unnecessary code that is never executed but confuses the reader when trying to understand the decompiled output.

- **Anti-debug**. Certain software engineers and hackers use debug tools to examine code line by line. With these tools, software engineers can detect problems with the code, and hackers can use them to reverse engineer the code. IT Security professionals can use anti-debug tools to detect when a hacker is running a debug program as part of an attack. Hackers can use anti-debug tools to detect when a to debug tool is being used to identify the changes they are making to the code.
- **Anti-tamper**. These tools detect code that has been tampered with, and if it has been modified, it stops the program.
- **String encryption**. This method uses encryption to hide the strings in the executable and restores the values only when they are needed to run the program. This makes it difficult to traverse a program and search for strings.
- **Code transposition**. This is the reordering of routines and branches in the code without having a visible effect on its behavior.

4.0 Implementation and analysis of obfuscated software

Code obfuscation is not about changing the content of the original code of a program, but rather about making the way that code is transmitted and presented more confusing. Obfuscation doe not change the manner in which the program works or its last result.

What follows is an example snippet of simple Golang code obfuscation:

Source code

```
package main

import "fmt"

func main() {
        fmt.Println("Hello world!")
}
```

Obfuscated code

```
package main

import "fmt"

func main() {
```

```
        fmt.Println((func() string {
                mask := []byte("\x21\x0f\xc7\xbb\x81\x86\x39\xac\x48\xa4\xc6\xaf")
                maskedStr :=
[]byte("\x69\x6a\xab\xd7\xee\xa6\x4e\xc3\x3a\xc8\xa2\x8e")
                res := make([]byte, 12)
                for i, m := range mask {
                        res[i] = m ^ maskedStr[i]
                }
                return string(res)
        }()))
}
```

Code 1 – Example of the obfuscated code when string obfuscation method implemented

Golang provides a great opportunity for malware developers to build tools that challenge malware researchers and reverse engineers while being inherently capable of compiling across different architectures. Meanwhile, some people have developed various tools to help with static reverse engineering of these binaries, furthermore, different tools have been created to display the best way to attack these procedures through different obfuscation strategies. Most current tools for static inversion of Golang binaries rely on the existence of the .gopclntab structure, which contains a listing of function names and their offsets even in stripped binaries, thus individuals can enumerate this structure and rename all stripped functions to their default library names. This structure is not needed and can be erased or counted by one more interaction to destroy the capacity names and create extra confusion. There are two methods of obfuscating compiled Golang binaries. One does this before it is compiled by manipulating the source code, the other can enumerate the .gopclntab structure of a previously assembled binary to control it.

During the implementation phase, I used the Garble to obfuscate the Golang code by wrapping the Go toolchain. The purpose of the Garble is to create a binary that works as well as a regular build but contains as little information about the source code as possible. The tool is designed to:

- Coupled with `cmd/go`, to support modules and build caching
- Deterministic and reproducible, given the same initial source code
- The give the source is reversible, to de-obfuscate panic stack traces

Garble wraps call to the Go compiler and linker to transform the Go build, in contemplation of:

- Replace as many useful identifiers as possible with short base64 hashes

- Replace package paths with short base64 hashes
- Remove all build and module information
- Strip filenames and shuffle position information
- Strip debugging information and symbol tables via `-ldflags="-w -s"`
- Obfuscate literals, if the `-literals` flag is given
- Remove extra information, if the `-tiny` flag is given

Analysis. In this experiment, the specified application was created using the Garble Obfuscation tool. Currently, no known tool decompiles Golang sources. Go is a local language compiler that, dissimilar to Java and comparative dialects that depend on a class loader, utilizes a linker to make its executables. Therefore, it is sufficient to reconstruct the call tree, as with C and similar languages. If you need to reconstruct a Go program, you are better off using traditional, less complex reverse engineering tools such as IDA Pro.

To examine the success of obfuscation, all steps are reproduced gradually from Identifying the problem.

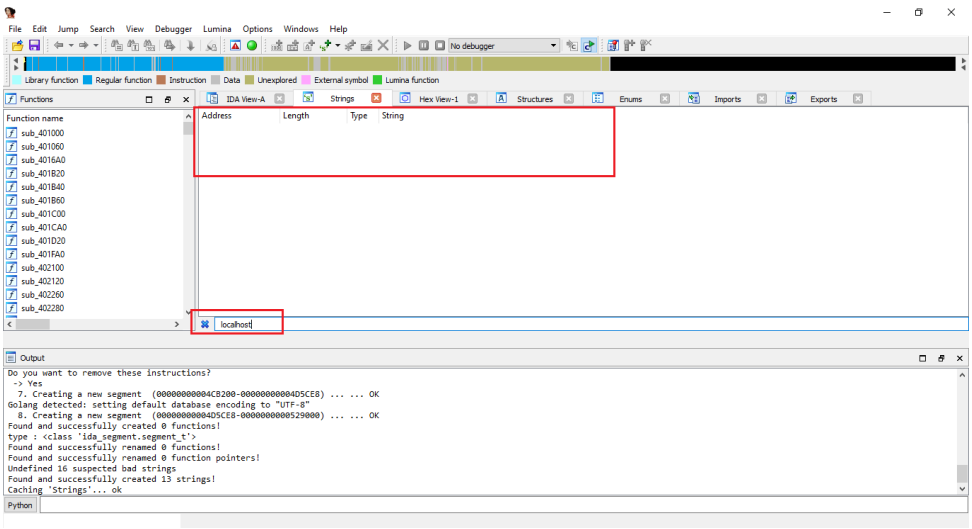

Figure 3 – String analysis on the IDA (Binary code analysis tool) when application obfuscated with garble.

The figure above displays the analysis of the obfuscated binary on the IDA. On the left-hand side, all renamed functions have been displayed. Moreover, on the right-hand side “String Finder”, view has been displayed. There is no “localhost” string in binary code (All strings are hidden by garble). In this case, the success of the obfuscation techniques has been successfully proven.

5.0 Conclusion

In conclusion, the main goal of this paper is to fully explain the meaning of software obfuscation. No one is trustworthy when it comes to software security, not even cloud providers. During the experiment, the binary software was modified and a malicious code was inserted on the IDA (binary code analysis tool) by the intruder. To ensure the integrity of the software, it is important to ensure that the source code is not tampered with. Finally, in the experiment, the source code was obfuscated using the Garble - Golang obfuscation tool. The results showed that it is impossible to understand it or apply reverse engineering patterns to it. However, since software protection and defense are highly interactive domains, the focus may shift and new research challenges may emerge.

References

- TechTarget, SearchSecurity. What is obfuscation and how does it work? https://www.techtarget.com/searchsecurity/definition/obfuscation
- Github, burrowers/garble, https://github.com/burrowers/garble
- Medium, De-obfuscating GoLang Functions, Jason Reaves, https://medium.com/walmartglobaltech/de-ofuscating-golang-functions-93f610f4fb76
- Binary Ninja Blog, Deobfuscation of Gobfuscate Golang Binaries Across Multiple Architectures, https://binary.ninja/2020/12/02/deobfuscation-of-gobfuscate-golang-binaries.html

Appendix

Github, Obfuscation Experiment,

https://github.com/eminmuhammadi/obfuscation-experiment

Figure 1 – https://raw.githubusercontent.com/eminmuhammadi/obfuscation-experiment/master/screenshots/01.PNG

Figure 2 – https://raw.githubusercontent.com/eminmuhammadi/obfuscation-experiment/master/screenshots/02.PNG

Figure 3 – https://raw.githubusercontent.com/eminmuhammadi/obfuscation-experiment/master/screenshots/04.PNG