Reverse Engineering

Accordi Gianmarco

1 Introduction

Reverse Engineering is the process of understanding how system made by man is working, by looking at it an trying to reverse the operations it does in order to understand its behavior. Its like a scientific research but in this case we are not interested in natural phenomenon. This kind of process has been applied to a lot of different fields, from mechanical engineering to chemical engineering, so as the name suggested it is applied to all field of engineering: these are the fields in which we can take apart the work done by other human being and trying to understand how they have build such a system. The necessity of this such an analysis of a system is usually required because creator of a system tends to disclose how they've built it, this mean that they obscure their work in order to protect their creators rights. The purpose of this document is to give an introduction on **Reverse Engineering** specifically in the field of Software Engineering. The process of Reverse Engineering applied to Software products can be done at any level of the production, but it usually involves two stages [1]:

- **Redocumentation**: since usually the given software product that we want to analyze is already compiled we want to be able to reach an higher level of abstraction to better understand the code;
- **Design Understanding**: once that we have been able to get an higher abstraction of our code we can use our capacity and knowledge in order to understand what the program does and how it does it, so we want to follow the development process of the creators of such code.

Disclose the design features can be understand with two approaches:

- **Dynamic Analysis**: this is an on the field approach, in which you launch the program and you register what the program is doing, and by analyzing the impact it has on the environment(print,systemcalls,...) you try to figure out what it does;
- **Static Analysis**: analyze the code without launch it, you analyze only the output you've obtained from the Redocumentation part.

In next sections we will analyze this approaches along with the stage involved in doing them. Then we proceed in order to define which is the best strategy based on what we have seen. Finally an example is provided taken from a CTF.

[1] `https://en.wikipedia.org/wiki/Reverse_engineering`

Reverse Engineering in Software Engineering As previously stated this document focus its attention on Reverse Engineering applied to software. When a new software is released it is usually provided already compiled to the specific architecture, so you get the binary of this code. The binary, as the name suggests, contains binary instructions that are targeted to be understood by a specific machine with a specific architecture, this means that it cannot be understand easily from a human being. For the developers this is usually a wanted feature, since this make the reverse engineering process more difficult for other companies that wants to disclose their design features. To be even more secure companies also rely on the usage of some Obfuscation Technique that makes the reverse engineering process even harder. This leads to the development of new research fields focused on trying to remove obfuscation, also in an automate way [5]. Some techniques used for obfuscate binaries are Packing(in which the source code pack and unpack the .text section as long as it execute)[2], Dynamic Code Mutation(in which the code mutate during execution), Code Generation(the program generates code during program execution) or by Binary Instumentation [4]. So in the end software is usually not fully disclose for two main reasons: protection of intellectual property and malware. In next sections we will see different techniques that can be used in order to defeat make the process of reverse engineering more easy.

2 Stages

The process of **Reverse Engineering** pass through 2 main stages, the Redocumentation part analysis statically the executable starting from a decompilation process. while the second part called Design Understanding involves the outcome of the previous stage combined with dynamic analysis, based on the strategy we choose to adopt.

2.1 Redocumentation

The program we get is usually a binary file so the first steps in order to get an higher level representation of it is to disassemble it. Figure 1 refers to the case

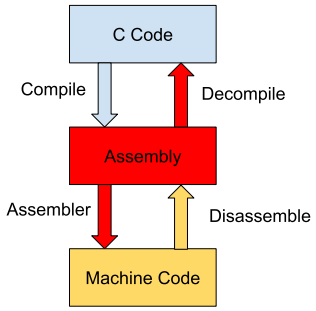

Figure 1: Compilation and Decompilation steps [3]

of compiled languages(and we will refers to the them from now on), as we can see when someone wants to make its code executable has to go throw a set of steps usually:

- **Compilation**: from the high level code(say for example C code) you use the compiler [2] to generate a low level language that usually is Assembly code;

[2] `https://en.wikipedia.org/wiki/Compiler`

- **Assembler**: in this stage you pass from Assembly code, that has an high correspondence with the machine code instructions so it is targeted to a specific architecture [3], into machine code that can be executed.

In fact what we usually have is the machine code, the executable file. From it we have to extract useful information to understand what it does, this again requires to revert the previous operations, as shown in 1(going down to up):

- **Disassemble**: from the machine code of a specific architecture(that can be x86-64 or arm) we get the correspondent Assembly code;
- **Decompile**: from the assembly code we have obtain our aim is to get the high level source code that has generate it as similar as possible to the original one.

The process of generating an executable is not an invertible function [4] This means that by doing so we lost some information, that makes the inverse process quite harder. In fact the output of the decompilation stage can vary based on the compiler used to perform compilation and the presence of metadata present inside of the executable, if for example has been included debug information or if instead it has been stripped of all the metadata. The outcome of the compilation change also based on the optimization enabled at compile time. As previously state some code can also be harder to be decompiled since it has been obfuscated with different techniques, another example is the insertion of instructions that doesn't change the flow of executed instructions but it insertion broke the alignment and so the decompiler is not able to recognize instructions in the correct way. The output of the Disassemble stage can be analyzed as shown in figure 2. We can organize block of assembly instructions inside block that are executed sequentially, until some instruction that controls the execution flow are encountered, like jump instruction. In such cases the flow is divided into different block of other assembly instructions. The output of the Decompilation part can be read mode easily since it will be something similar to the original source code that has produced the executable file. The readability of the output depends on the quality of the tool used to decompile and on the present metadata, but it is more than recommended a user inspection of code, for example to better understand the type of the variable. As you can see in image 4 the code on the right is more readable after inspection if the code, and the definition of a strut hat better fits the memory structure that is used by the executable. Up to know we have assumed to have under analysis an executable from a *Compiled Language*, but the same things can be applied to *Interpreted Language* [5] like for example Java, in which some tool exists in order to analysis the bytecode produced by java [6]. The work done during this stage is important in the first strategy that we will purse in order to discover design features of the programs under analysis: Static Analysis.

[3] `https://en.wikipedia.org/wiki/Assembly_language`
[4] `https://en.wikipedia.org/wiki/Decompiler`
[5] `https://en.wikipedia.org/wiki/Compiled_language`
[6] `http://java-decompiler.github.io/`

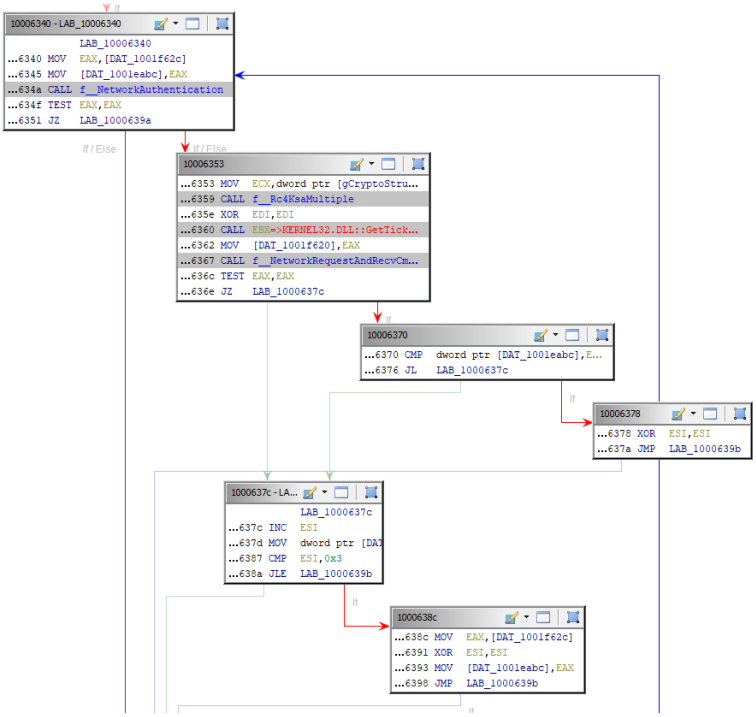

Figure 2: Analysis of the Decompilation's output stage.

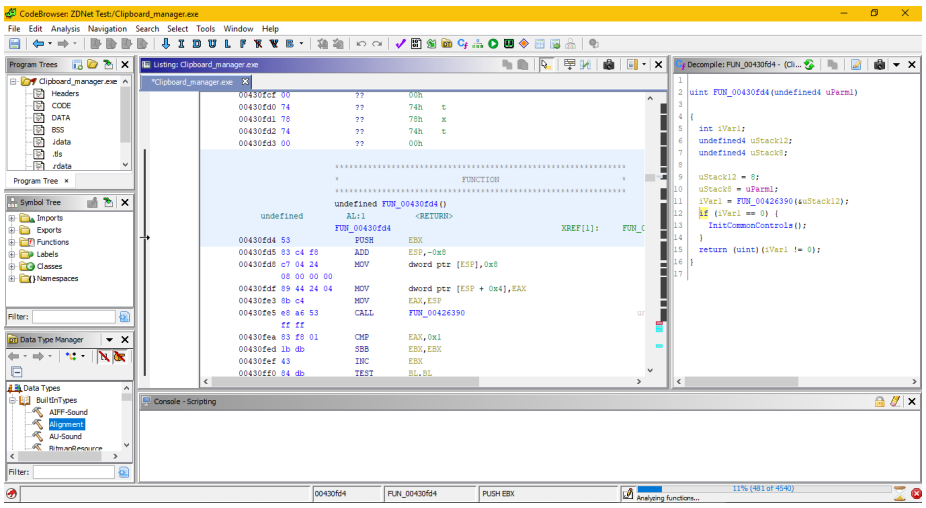

Figure 3: Ghidra GUI after analyzing an executable file.

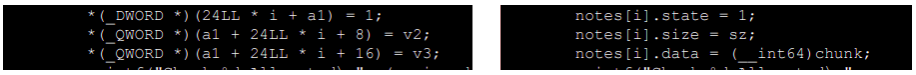

Figure 4: Code readability after inspection of the struct.

2.2 Design Understanding

The objective of the Reverse Engineering task is to understand the *Design Features* of the program under analysis. So in order to reach this results one can use different techniques, that can be generalized into two main categories: **Static Analysis** and **Dynamic Analysis**.

Static Analysis What we have seen during the *Redocumentation* stage 2.1 is what we can think as Static analysis, in fact it is the process of code analysis without regard of its execution or input [1]. From this analysis we can take a look at the:

- **Control flow**: the code is broke down into block, that merged together instructions that will be executed sequentially, this separation in block can be done by looking at how the control instructions are placed, like if or while loop 2, in this way we can try to understand which is the path block executed by the program sequentially, so we can follow the execution flow;
- **Data Flow**: helps in understanding how the data flows inside the program, from the input we can give to the program to the final outcome of its execution.

//TODO

Dynamic Analysis Opposed to Static the Dynamic Analysis instead rely on the execution of the program regarding on the input provided to it. In this case we try to launch the executable with different inputs and look at the different outcome we got. And by looking at the output returned by the executable we try

```
root@server1:~# strace -i ls
[00007f9ba2960a07] execve("/usr/bin/ls", ["ls"], 0x7ffdd2da3038 /* 19 vars */) = 0
[00007f8f9466c7b7] brk(NULL)             = 0x55732a3e8000
[00007f8f9466d417] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
[00007f8f9466d4fd] openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[00007f8f9466d383] fstat(3, {st_mode=S_IFREG|0644, st_size=33427, ...}) = 0
[00007f8f9466d6f3] mmap(NULL, 33427, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8f9464a000
[00007f8f9466d437] close(3)              = 0
[00007f8f9466d4fd] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
[00007f8f9466d5c4] read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@k\0\0\0\0\0\0"..., 832) = 832
[00007f8f9466d383] fstat(3, {st_mode=S_IFREG|0644, st_size=155296, ...}) = 0
[00007f8f9466d6f3] mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8f94648000
[00007f8f9466d6f3] mmap(NULL, 2259632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8f94420000
[00007f8f9466d7a7] mprotect(0x7f8f94445000, 2093056, PROT_NONE) = 0
[00007f8f9466d6f3] mmap(0x7f8f94644000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x7f8f94644000
[00007f8f9466d6f3] mmap(0x7f8f94646000, 6832, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f8f94646000
[00007f8f9466d437] close(3)              = 0
[00007f8f9466d4fd] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[00007f8f9466d5c4] read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260A\2\0\0\0\0\0"..., 832) = 832
[00007f8f9466d383] fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
[00007f8f9466d6f3] mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8f9425f000
[00007f8f9466d7a7] mprotect(0x7f8f94281000, 1658880, PROT_NONE) = 0
[00007f8f9466d6f3] mmap(0x7f8f94281000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7f8f94281000
[00007f8f9466d6f3] mmap(0x7f8f943c9000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7f8f943c9000
[00007f8f9466d6f3] mmap(0x7f8f94416000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000) = 0x7f8f94416000
[00007f8f9466d6f3] mmap(0x7f8f9441c000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f8f9441c000
[00007f8f9466d437] close(3)              = 0
[00007f8f9466d4fd] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
[00007f8f9466d5c4] read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\340!\0\0\0\0\0\0"..., 832) = 832
[00007f8f9466d383] fstat(3, {st_mode=S_IFREG|0644, st_size=469248, ...}) = 0
[00007f8f9466d6f3] mmap(NULL, 471528, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8f941eb000
[00007f8f9466d6f3] mmap(0x7f8f941ed000, 335872, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f8f941ed000
[00007f8f9466d6f3] mmap(0x7f8f9423f000, 122880, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x54000) = 0x7f8f9423f000
[00007f8f9466d6f3] mmap(0x7f8f9425d000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x71000) = 0x7f8f9425d000
[00007f8f9466d437] close(3)              = 0
[00007f8f9466d4fd] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
[00007f8f9466d5c4] read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000\21\0\0\0\0\0\0"..., 832) = 832
[00007f8f9466d383] fstat(3, {st_mode=S_IFREG|0644, st_size=14592, ...}) = 0
```

Figure 5: The possible output of strace after an executable execution.

to guess what the program does. We can also use some other tools to understand what the executable does by looking at the system calls invocation done by it, in this case we can user *strace*[7]. As we can see in figure 5 the output will make a summary of the various system calls done by a certain program. *Strace* allows to analysis how the executable access to the privileged kernel mode, how the program switch its execution from user mode to kernel mode. Other tools that can be used are *ltrace* and *ptrace*.

//TODO introduce debugger

//TODO make another section with an image with the relation between static and dynamic analysis, and how to use them togheter

2.3 Example of code

Here you can fine an example of how to add some code. It is using the package minted with basic option enabled. You can write code in place:

```
1  import numpy as np
2
3  def incmatrix(genl1,genl2):
4      m = len(genl1)
5      n = len(genl2)
6      M = None #to become the incidence matrix
7      VT = np.zeros((n*m,1), int)  #dummy variable
8
9      #compute the bitwise xor matrix
10     M1 = bitxormatrix(genl1)
11     M2 = np.triu(bitxormatrix(genl2),1)
12 ...
```

You can reference to a line The important line is line 5.

You can use an external file and ave the code as listing.

The important line is line 3 of Listing 1.

3 Conclusion

References

[1] Liam O'Brien. *Reverse Egineering Lesson.*

[2] Mario Polino. *Packers and Evasive Techniques Slide.*

[3] Mario Polino. *Reverse Egineering Slide.*

[4] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. *Measuring and Defeating Anti-Instrumentation-Equipped Malware.* 2017.

[7] `https://en.wikipedia.org/wiki/Strace`

```
import numpy as np

def incmatrix(genl1,genl2):
    m = len(genl1)
    n = len(genl2)
    M = None #to become the incidence matrix
    VT = np.zeros((n*m,1), int)  #dummy variable

    #compute the bitwise xor matrix
    M1 = bitxormatrix(genl1)
    M2 = np.triu(bitxormatrix(genl2),1)
```

Listing 1: Example from external file

[5] S. K. Udupa, S. K. Debray, and M. Madou. *Deobfuscation: reverse engineering obfuscated code*. 2005.