National Institute Of Science And Technology Of Tunisia

Reverse Engineering Fundamentals

Authors :

Name and Forename :

Med Ali Ouachani

Iyed Mejri

Year :
2022 - 2023

I - Reverse Engineering Motivation :

We are not going to complicate things for you, for the sake of discussion, we loosely define reverse engineering as the process of understanding a system. It is a problem-solving process. A system can be a hardware device, a software program, a physical or chemical process, and so on. The process consists of trying to either rebuild stuff from zero or try to understand how things behave such as malicious programs referred to as malware, bypassing security that was built to protect the software, or maybe breaking a cheating engine for video games in which Anti-Cheat Engineer's exist, or maybe protecting them from getting cracked. Now, why should someone start learning this? clearly first, it's because this kind of skill is highly required nowadays, not everyone can do this it takes time to master it, no one can call himself a master of reverse engineering until he has truly proved his ability to explain the abstract theory's behind computers and systems, there are few people who can understand truly how a machine behave and that's our goal to understand it, so that we can exploit it. At the end, if you take a look at the reverse engineer's salaries, they are super high compared to software engineers.

II - Pre-requirement :

1 - Installing Linux operating system :

First of all, you need to install Linux operating system. Everything that we will be dealing with in the workshop will be involving that operating system.

You can install Windows Subsystem for Linux, referred to as WSL from (https://ubuntu.com/wsl) which contains only the terminal. If you want a graphic user interface feel free to download the full operating system image from (https://ubuntu.com/download/desktop) and install it on a virtual machine.

2 - Having a decent understanding of programming languages :

Once you installed Linux, fire it up and take a seat. In reverse engineering, the language is the architecture and assembly language. A word is an assembly instruction. Paragraphs are sequences of assembly instructions. A book is a program. However, to fully understand a book, the reader needs to know more than just vocabulary and grammar. These additional elements include structure and style of prose, unwritten rules of writing, and others. Understanding computer programs also requires a mastery of concepts beyond assembly instructions. Don't be afraid if you don't know what assembly Language means it will be covered in the next workshops. we would be misleading you if we were to claim that reverse engineering is a simple learning endeavor and that it can be completely mastered from this workshop. The learning process is quite involved because it requires knowledge from several disparate domains of knowledge. For example, an effective reverse engineer needs to be knowledgeable in computer architecture, systems programming, operating systems, compilers, and so on. For certain areas, a strong mathematical background is necessary. The system that we will be working on is a software program. To understand a program, you must first understand how software is written. Hence, the first requirement is knowing how to program a computer through a language such as C, C++, Java, and others. We suggest first learning C due to its simplicity and effectiveness.

3 - Binary/Hexadecimal number systems :

Binary numbers are composed of 2 numbers only 0's and 1's, each number in that system needs to be represented as the sum of powers of 2. I love examples, let's take one. We wanna know the equivalent of the number 14 in the binary system. 14 can be written as $14 = 1*2^3+1*2^2+1*2^1+0*2^0$. Which can be written in the binary system as $0b1110$. There is a little remark, the "0b" stands for binary number and the most significant bit is always the left-most (MSB) one, and the least significant bit is the right-most one (LSB).

Hexadecimal numbers are composed of 16 numbers, 10 of which are the digits from $0-9$ and the rest are the charterers from A - F. Each number in that system needs to be represented as the sum of powers of 16. Okay let's take a quick example, 14 can be written in the hexadecimal system as $14 = 14 * 16^0$. As you can see 14 is above 9 so we need to find the character equivalent to 14, it's "E" which means that 14 is written in the hexadecimal system as 14 = 0xE. A little remark, "0x" stands for hexadecimal, to convert a bigger number to hexadecimal it's easier to convert it first to a binary number and then divide

it into blocs of 4 and then convert each bloc to hexadecimal. An example, $27 = 0b11011$. We divide the binary representation into blocs of 4, we get $27 = 0b00011011$. The first block $0b0001 = 1 = 0x1$, the second block $0b1011 = 11 = 0xB$. So the number 27 = 0x1B in hexadecimal.

There is a simple way to convert numbers to binary and hexadecimal using python, or by writing an algorithm, it's an easy task to do.

```
x = 27
print("The binary representation of 27 : ", bin(27))
# Result would be '0b11011'
print("The Hex representation of 27 : ", hex(27))
# Result would be '0x1b'
```

4 - Bitwise Operations :

a - AND operation :

The AND bitwise operation is applied between 2 bits, it gives (true = 1) when both of the bits are equal to 1 else it gives (false = 0).

A table can explain that more clearly:

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

b - OR operation :

The OR bitwise operation is applied between 2 bits, it gives (true = 1) when one of the bits is equal to 1 or both else it gives (false = 0).

A table can explain that more clearly:

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

c - NOT operation :

The NOT bitwise operation is applied to a bit, it gives the opposite of the bit that's been given to it.

A table can explain that more clearly:

X	NOT(X)
0	1
1	0

d - XOR operation :

The XOR bitwise operation is applied between 2 bits, it gives (true = 1) when one of the bits is equal to 1 else it gives (false = 0).

A table can explain that more clearly:

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

e - Shifting operations :

When it comes to shifting operations, it's divided into 2 operations. The first one is right-shifting. Let's take an example, let's right-shift 10($0b1010$) by 2. First shift gives $0b0101$, second shift gives $0b0010$. So as you can see, after 2 right shifts the number $0b1010 = 10 => 0b0010 = 2$. What that right-shifting was doing is actually dividing the number by 2 in each shift he is been doing. Meanwhile, left-shifting is the opposite of that, it's multiplying by 2 in each shift you do. We take an example, let's left-shift the number 2 by 4. $2 = 0b00010 => 4 = 0b00100 => 8 = 0b01000 => 16 = 0b10000$. So left-shifting of $2 = 16$. Nice, I'm pretty sure you understand shifting well now.

This operation is already implemented in C and other languages :

```c
int x = 2;
// this a left shift by 2
x = x << 2;
// now, x = 8
// this a right shift by 3
x = x >> 3;
// now, x = 1
```

f - Rotation operations :

Simply, Bit Rotation: A rotation (or circular shift) is an operation similar to a shift except that the bits that fall off at one end are put back to the other end.

In left rotation, the bits that fall off at left end are put back at right end. An example, Let n is stored using 8 bits. Left rotation of n = 11100101 by 3 makes n = 00101111 (Left shifted by 3 and the first 3 bits are put back in last). If n is stored using 16 bits or 32 bits then left rotation of n (000...11100101) becomes 00..0011100101000.

In right rotation, the bits that fall off at right end are put back at left end. An example, Right rotation of n = 11100101 by 3 makes n = 10111100 (Right shifted by 3 and the last 3 bits are put back in first) if n is stored using 8 bits. If n is stored using 16 bits or 32 bits then right rotation of n (000...11100101) by 3 becomes 101000..0011100.

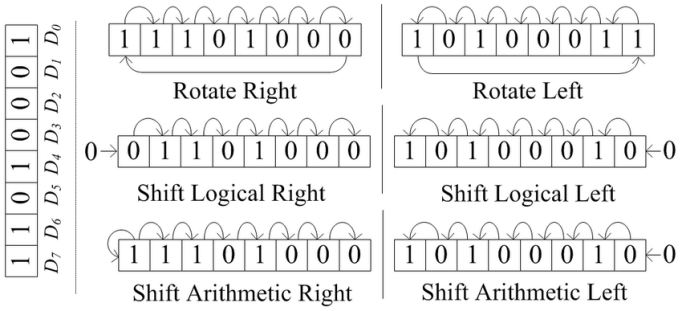
This operation is not defined in the C, so we should write it ourselves:

```c
#define INT_BITS 32

int leftRotate(int n, unsigned int d) {
   return (n << d)|(n >> (INT_BITS - d));
}

int rightRotate(int n, unsigned int d) {
   return (n >> d)|(n << (INT_BITS - d));
}
```

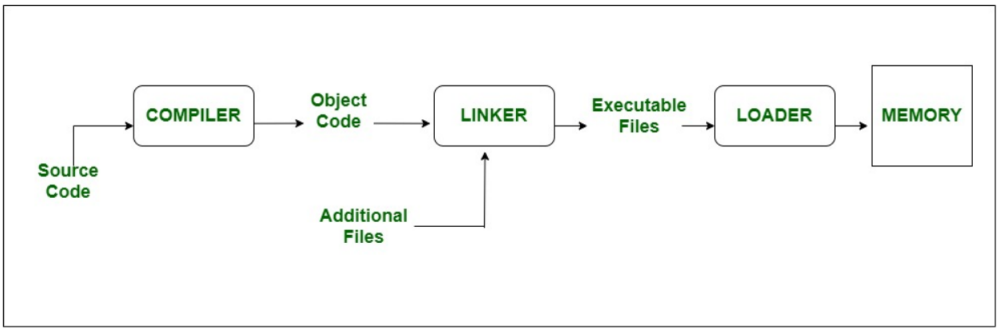
A picture that sums up all the shift/rotate operations :



IV - Compilers, Linkers and Loaders:

In this part, we will try to explain the execution process of a binary in the computer briefly.

First, you start by writing your code in the programming language that you want to use. Then the compiler job comes into action to compile it, the compiler is simply a piece of software that's been written with the goal of translating a programming language's source code into machine code, bytecode, or another programming language. After the compilation process is over, we have got an object code that can't be executed directly by the CPU, that's why we need to link libraries and stuff to it, and here comes the job of the linker, to link those libraries and generate an executable which is also referred as a binary. What resets is now is to load that file in the memory, here comes a piece of software called loader that's been included with the operating system, his job is to load that file into memory and let the CPU execute it.

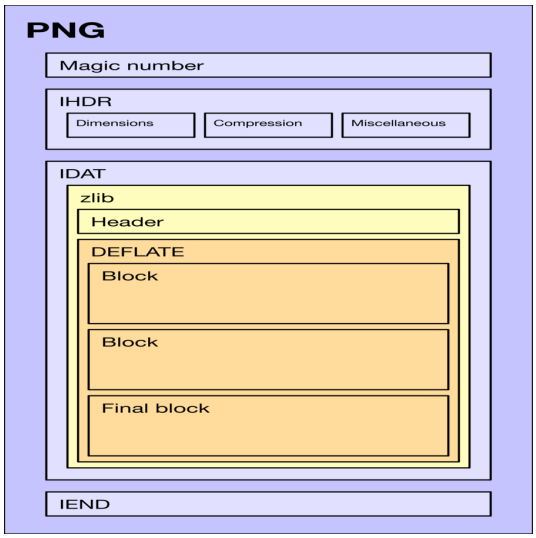


Here's a graph that can describe the process briefly!

IV - Executable format ELF/PE:

1 - File format :

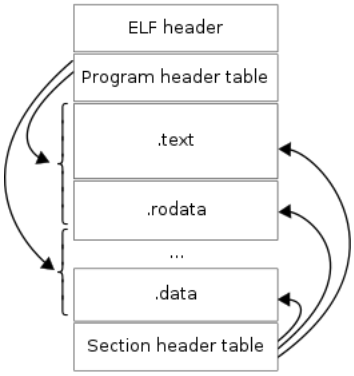
We all know the importance of files and file formats, especially on Linux where everything is a file. Every file has a certain structure related and specific to it that let programs know how to manipulate it (for example read data from it or write). ELF *Executable and Linkable Format* defines the structure for executables, libraries, etc. on Unix/Linux systems. On the other hand, PE *Portable Executable* defines the structure for executables, DLLs, etc. on Windows systems. First I would suggest taking a look at the PNG file structure as it will give a better understanding of the concept of file formats.



Every file format starts with a magic number also called the file signature, which are constant bytes reserved for every file. For example the PNG file signature is **\x89PNG\x0d\x0a\x1a\x0a**. After the signature each file has it's own way of representing it's structure, the PNG first introduces a chunk of bytes called the IHDR (Image Header) containing the width, height, filter... Then the PNG format will initializes a chunk or more of the IDAT (Image Data) which contains all of the image's compressed pixel data. At the end of the PNG structure there is a string *IEND* marking the end of the structure. Now having a basic idea of what are file formats let's look at the ELF and PE formats.

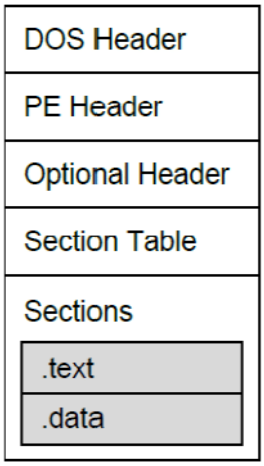
2 - ELF format :

Identical to the PNG format the ELF format starts with it's own signature which is **\x7fELF**, then it starts by initializing the numbers of bits of an the architecture the program targets, the endianness, the address where the executable starts its execution from called *entry point*... The program header table contains data describing segments (one or more section) and other information needed to prepare for execution. The section header table contains informations describing each of the file's sections (address, name, size...).



3 - PE format :

Now looking at the PE file format, the PE file starts also with it's own signature *MZ* (starting at the DOS Header) just like any other file. The Dos Header is containing information making sure that the file is a valid PE. The PE Header and the Optional Header contains information related to the properties of the file (the Machine it is supposed to run on, Number of sections, Number of symbols, The size of all the sections...) The section table contains informations about each of the sections that are present (name, size...).



You might ask on, why are we even needing to learn this formats for? Well, it helps to learn the inner workings of our operating system or when something goes wrong we might better understand what happened (or why).

V - Static/Dynamic Analysis :

- Dynamic analysis is the process in which we try to get information about the application and the way its working just by interacting with it during runtime without reading any code.
- Static analysis is the process in which we try to get information about the application by just reading its code without interacting with it.
- In dynamic analysis we could use tools like debuggers / tracers to help us get knowledge about some functionalities.
- It's always preferable (in some cases) to start doing the dynamic analysis before the static analysis as this would make you know how to walk through the code easily, making the static analysis faster and better and making you know exactly what points you should look at.
- Of course static analysis is really important and should always be present with dynamic analysis for a good reverse engineering of a system.

VI - Challenges :

1 - First Challenge :

This is a piece of code, could you please reverse it for me and retrieve the flag?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*################
Authors => IronByte X t0m7r00z
################*/

int main(int argc, char* argv[]) {
        int t[36] = {5, 141, 43, 73, 114, 50, 197, 26, 166,
                     235, 37, 150, 75, 115, 4, 82, 35, 235,
                     201, 21, 198, 220, 162, 8, 184, 40, 239,
                     37, 49, 97, 177, 4, 175, 13, 197, 138};
        char input[0x30];

        srand(2022);
        printf("Give me the flag please: ");
        scanf("%48s", input);
        if (strlen(input) != 36)
                printf("Wrong Length!");
        else {
                int i = 0;
                int flag = 1;
                while(i < 36 && flag) {
                        int nb = rand();
                        int x = (uint)(nb >> 31) >> 24;
                        x = (nb + x & 0xff) - x;
                        x = x ^ t[i];
                        if (input[i] != x) {
                                flag = 0;
                                break;
                        }
```

```
                        i++;
                }
                if (!flag)
                        printf("Wrong flag!");
                else
                        printf("Nice you can validate with that flag!");
        }
        return 0;
}
```

To simply Reverse that code we need first to input a string with length 36 to bypass the first check, second thing to notice was that "int nb = rand()", which is generating a random number, the question is how do we generate the same numbers like him, the third thing to notice was that he is applying some bitwise operations on that number that he is xoring that value with values from the array above. The length problem can be solved quickly by just making sure that our flag length is 36. The second problem which generates the same random numbers that he has been generating can be simply solved by finding the seed that he has been using along the code which is equal to 2022 in our case then we xor the i-th number we generated with the i-th element from the array above. That's it, let's write the code that can reverse that piece of code.

This is a C Solver :

```c
#include <stdio.h>
#include <stdlib.h>

/*################
Authors => IronByte X t0m7r00z
################*/

/* flag = Securinets{Ign0r3_th4T_jUnK_4nD_XOR} */

int main(int argc, char* argv[]) {
        int t[36] = {5, 141, 43, 73, 114, 50, 197, 26, 166,
                     235, 37, 150, 75, 115, 4, 82, 35, 235,
                     201, 21, 198, 220, 162, 8, 184, 40, 239,
                     37, 49, 97, 177, 4, 175, 13, 197, 138};
        srand(2022);
        printf("Flag: ");

        for (int i = 0; i < 36; i++) {
                int nb = rand();
                int x = (uint)(nb >> 31) >> 24;
                x = (nb + x & 0xff) - x;
                x = x ^ t[i];
                printf("%c", x);
        }

        puts("");
        return 0;
}
```

This is a python Solver:

```python
#!/usr/bin/python3

from ctypes import CDLL

libc = CDLL("./libc.so.6")
t = [5, 141, 43, 73, 114, 50, 197, 26, 166, 235, 37, 150,
75, 115, 4, 82, 35, 235, 201, 21, 198, 220, 162, 8, 184,
40, 239, 37, 49, 97, 177, 4, 175, 13, 197, 138]
libc.srand(2022)
flag = bytearray(len(t))

for i in range(len(t)):
    flag[i] = (libc.rand() & 0xff) ^ t[i]

assert(flag == b"Securinets{Ign0r3_th4T_jUnK_4nD_X0R}")
print(flag.decode("UTF-8"))

```

2 - Second Challenge :

Code of the challenge :

```c
#include <stdio.h>
#include <string.h>

/*################
Authors => IronByte X t0m7r00z
################*/

unsigned short rol16(unsigned short x, unsigned int bits) {
        return (x << bits) | (x >> (16 - bits));
}

int main(int argc, char* argv[]) {
        char input[0x50];

        printf("Give me the flag please: ");
        scanf("%64s", input);
        if (strlen(input) != 53)
                puts("Wrong Length!");
        else {
                for (int i = 0; i < 53; i++) {
                        unsigned short *psh = &input[i];
                        if (i & 1)
                                *psh = rol16((unsigned short)*psh, 8);
                }
                if (!strcmp(input, "Scerunite{s4p1ldN0r3M&_1_1__
  s0CCo_L5196744813486543}1"))
```

```
                        puts("Well Done!");
                else
                        puts("Not really what I was waiting for :(");
        }
        return 0;
}
```

C Solver :

```
#include <stdio.h>

unsigned short rol16(unsigned short x, unsigned int bits) {
        return (x << bits) | (x >> (16 - bits));
}

int main() {
        char input[0x50] = "Scerunite{s4p1ldN0r3M&_1_1__s0COo_L5196744813486543}1\0";

        for (int i = 0; i < 53; i++) {
                unsigned short *psh = &input[i];
                if (i & 1)
                        *psh = rol16((unsigned short)*psh, 8);
        }
        printf("%s\n", input);
        return 0;
}
```