Reverse Engineering

Accordi Gianmarco

1 Introduction

Reverse Engineering is the process of understanding how a 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. Such an analysis is usually required because creator of a system tends to do not 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, and 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 reconstruct the original source 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, by analyzing the impact it has on the environment(print,systemcalls,...);
- **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 for a specific architecture, so you have to start your analysis from a binary. 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 [6]. 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 these obfuscation methods, that makes 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.

2.1 Redocumentation

As said, the program we get is usually a binary file so the first step in order to get an higher level representation of it is to disassemble it. Figure 1 refers to the case of compiled languages(and we will refers to the them from now on), the image highlight different operations that has to be done in order to go frm the source code to the binary(from top to bottom):

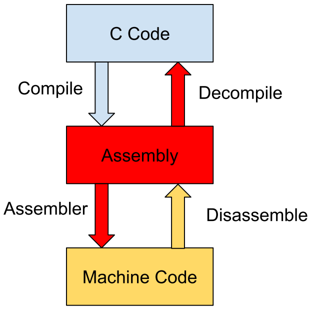


Figure 1: Compilation and Decompilation steps [3]

- **Compilation**: from the high level code(say for example C code) you use the compiler [2] to generate a lower level set of instructions, 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(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(from bottom to top):

- **Disassemble**: from the machine code of a specific architecture(that can be x86-64 or arm) we get the correspondent Assembly code;

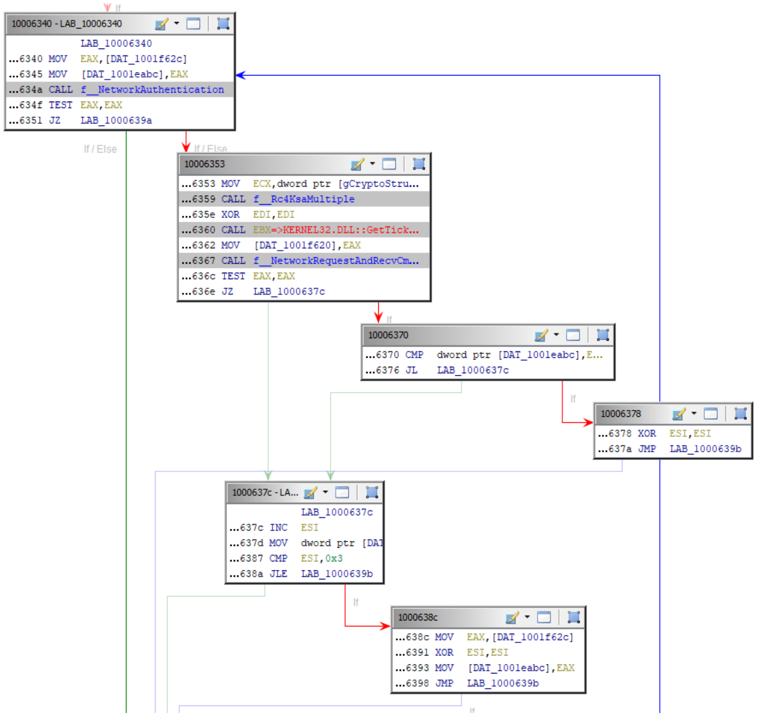
- **Decompile**: from the assembly code we want to obtain a high level code that has generate it, as similar as possible to the original one.

The process of generating an executable is not an invertible function [4] This means that by doing so we loose 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, the optimizations enabled(like -O in the gcc compiler) 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. 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 execution flow of instructions, but their 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 instructions. 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. Different tools exists that allows to perform decompilation of an executable. Most famous one are *Ghidra* and *IDA Pro*. Figure 3 show an example output of Ghidra. The reference of supported architecture for decompilation can be found here [5]. Ghidra makes use of automated tools that helps in generating the source code starting from the executable. From interface we can look at the whole space of the object file: from the address of the library, to the assembly code, to the constant variables loaded inside the 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

[3] `https://en.wikipedia.org/wiki/Assembly_language`

[4] `https://en.wikipedia.org/wiki/Decompiler`

[5] `https://en.wikipedia.org/wiki/Ghidra`



```
LAB_10006340
...6340 MOV  EAX,[DAT_1001f62c]
...6345 MOV  [DAT_1001eabc],EAX
...634a CALL f__NetworkAuthentication
...634f TEST EAX,EAX
...6351 JZ   LAB_1000639a
```

```
...6353 MOV  ECX,dword ptr [gCryptoStru...
...6359 CALL f__Rc4KsaMultiple
...635e XOR  EDI,EDI
...6360 CALL EBX=>KERNEL32.DLL::GetTick...
...6362 MOV  [DAT_1001f620],EAX
...6367 CALL f__NetworkRequestAndRecvCm...
...636c TEST EAX,EAX
...636e JZ   LAB_1000637c
```

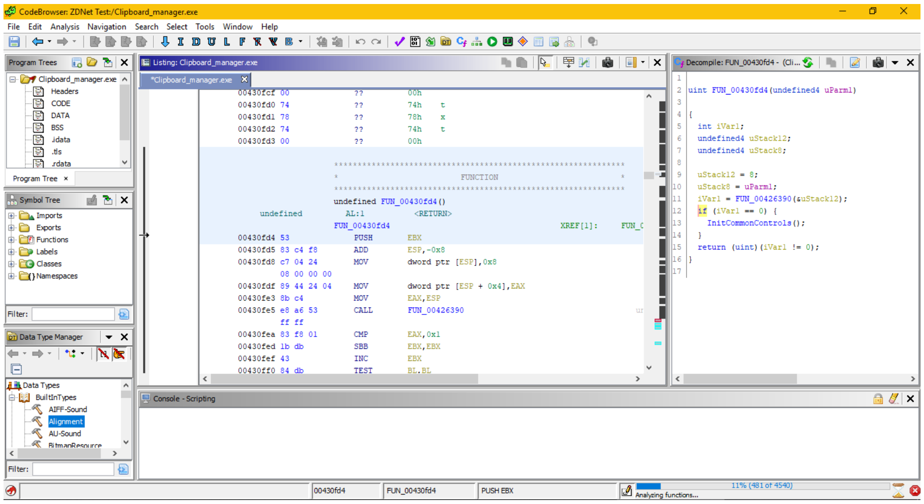
```
...6370 CMP  dword ptr [DAT_1001eabc],E...
...6376 JL   LAB_1000637c
```

```
...6378 XOR  ESI,ESI
...637a JMP  LAB_1000639b
```

```
             LAB_1000637c
...637c INC  ESI
...637d MOV  dword ptr [DAT
...6387 CMP  ESI,0x3
...638a JLE  LAB_1000639b
```

```
...638c MOV  EAX,[DAT_1001f62c]
...6391 XOR  ESI,ESI
...6393 MOV  [DAT_1001eabc],EAX
...6398 JMP  LAB_1000639b
```

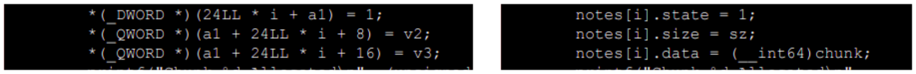
Figure 2: Analysis of the Decompilation's output stage.



```
00430fcf 00              ??          00h
00430fd0 74              ??          74h    t
00430fd1 78              ??          78h    x
00430fd2 74              ??          74h    t
00430fd3 00              ??          00h

                   **************************************************************
                   *                          FUNCTION                          *
                   **************************************************************
                   undefined FUN_00430fd4()
     undefined       AL:1          <RETURN>
                   FUN_00430fd4                                 XREF[1]:     FUN_(
00430fd4 53              PUSH        EBX
00430fd5 83 c4 f8        ADD         ESP,-0x8
00430fd8 c7 04 24        MOV         dword ptr [ESP],0x8
         08 00 00 00
00430fdf 89 44 24 04     MOV         dword ptr [ESP + 0x4],EAX
00430fe3 8b c4           MOV         EAX,ESP
00430fe5 e8 a6 53        CALL        FUN_00426390
         ff ff
00430fea 83 f8 01        CMP         EAX,0x1
00430fed 1b db           SBB         EBX,EBX
00430fef 43              INC         EBX
00430ff0 84 db           TEST        BL,BL
```

```

uint FUN_00430fd4(undefined4 uParm1)

{
  int iVar1;
  undefined4 uStack12;
  undefined4 uStack8;

  uStack12 = 8;
  uStack8 = uParm1;
  iVar1 = FUN_00426390(&uStack12);
  if (iVar1 == 0) {
    InitCommonControls();
  }
  return (uint)(iVar1 != 0);
}

```

Figure 3: Ghidra GUI after analyzing an executable file.



```
*(_DWORD *)(24LL * i + a1) = 1;
*(_QWORD *)(a1 + 24LL * i + 8) = v2;
*(_QWORD *)(a1 + 24LL * i + 16) = v3;
```

```
notes[i].state = 1;
notes[i].size = sz;
notes[i].data = (__int64)chunk;
```

Figure 4: Code readability after inspection of the struct.

under analysis an executable from a *Compiled Language*, but the same things can be applied to *Interpreted Language* [6] like for example Java, in which some tool exists in order to analysis the bytecode produced by java [7]. 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.

[6] https://en.wikipedia.org/wiki/Compiled_language

[7] http://java-decompiler.github.io/

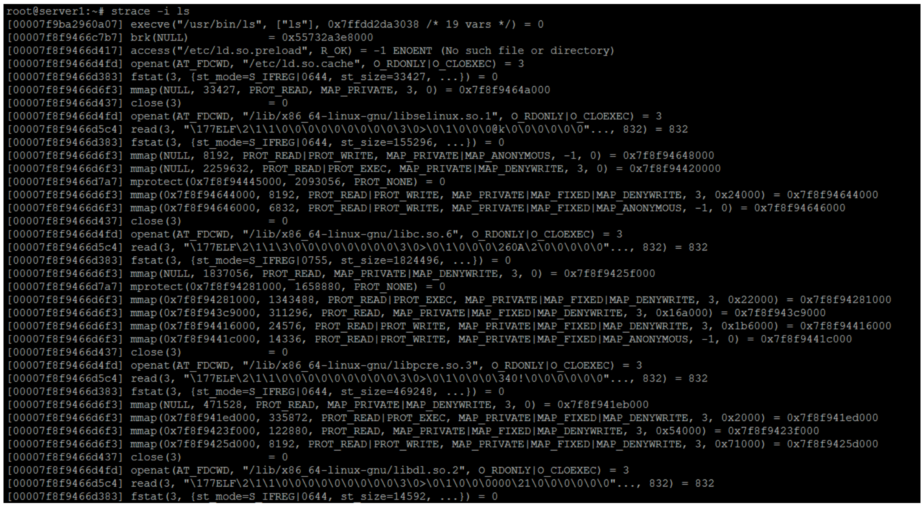
2.2 Design Understanding

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 blocks, that merge together instructions 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 that we can give to the program to the final outcome of its execution.

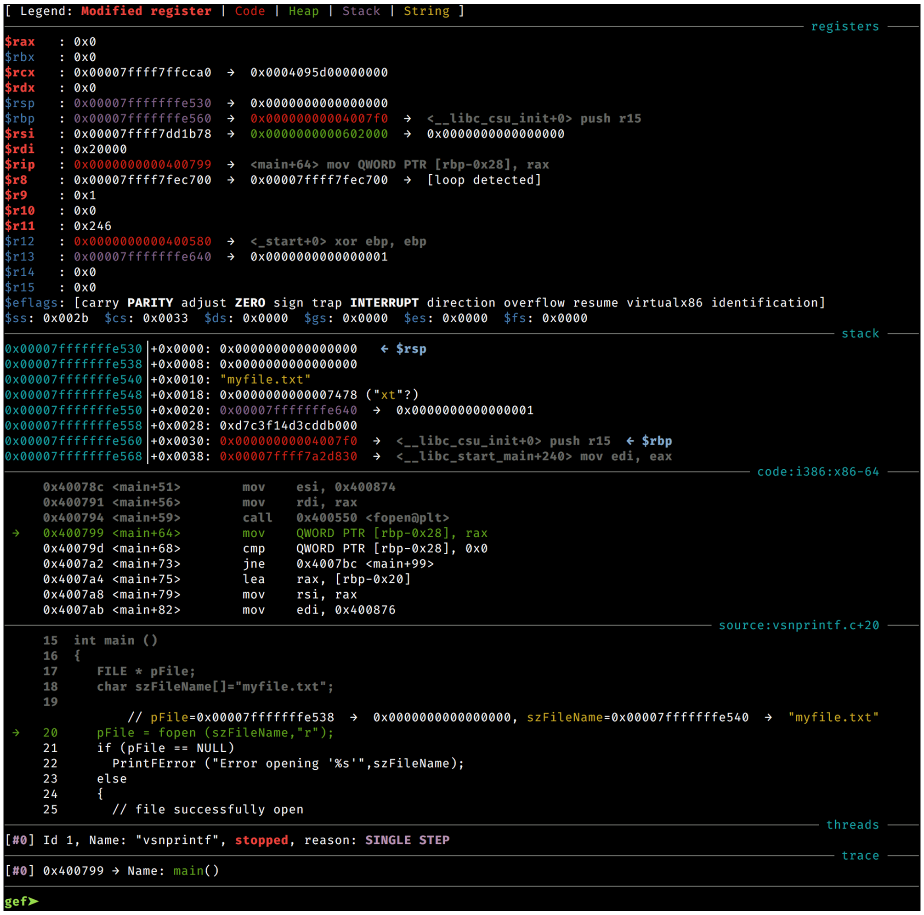
Dynamic Analysis Opposed to Static, Dynamic Analysis rely on the execution of the program regarding on the input provided to it. In this case we try to launch the executable with different inputs and look at the different outcome we got. And by looking at the output returned by the executable we try to guess what the program does. We can also use some other tools to understand



```
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 output of an analysis with strace.

what the executable does by looking at the system calls invocation done by it, in this case we can use *strace*[8]. 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*. In general you can look to the access made by the program to the disk memory, to the network usage of the program, etc.... Since during Dynamic Analysis we have to be able to analyze the behavior of the



```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers ──
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x00007ffff7ffcca0  →  0x0004095d00000000
$rdx   : 0x0
$rsp   : 0x00007fffffffe530  →  0x0000000000000000
$rbp   : 0x00007fffffffe560  →  0x00000000004007f0  →  <__libc_csu_init+0> push r15
$rsi   : 0x00007ffff7dd1b78  →  0x0000000000602000  →  0x0000000000000000
$rdi   : 0x20000
$rip   : 0x0000000000400799  →  <main+64> mov QWORD PTR [rbp-0x28], rax
$r8    : 0x00007ffff7fec700  →  0x00007ffff7fec700  →  [loop detected]
$r9    : 0x1
$r10   : 0x0
$r11   : 0x246
$r12   : 0x0000000000400580  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe640  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]
$ss: 0x002b  $cs: 0x0033  $ds: 0x0000  $gs: 0x0000  $es: 0x0000  $fs: 0x0000
──────────────────────────────────────────────────────────────────────────── stack ──
0x00007fffffffe530│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffe538│+0x0008: 0x0000000000000000
0x00007fffffffe540│+0x0010: "myfile.txt"
0x00007fffffffe548│+0x0018: 0x0000000000007478 ("xt"?)
0x00007fffffffe550│+0x0020: 0x00007fffffffe640  →  0x0000000000000001
0x00007fffffffe558│+0x0028: 0xd7c3f14d3cddb000
0x00007fffffffe560│+0x0030: 0x00000000004007f0  →  <__libc_csu_init+0> push r15  ← $rbp
0x00007fffffffe568│+0x0038: 0x00007ffff7a2d830  →  <__libc_start_main+240> mov edi, eax
─────────────────────────────────────────────────────────────────── code:i386:x86-64 ──
     0x40078c <main+51>        mov    esi, 0x400874
     0x400791 <main+56>        mov    rdi, rax
     0x400794 <main+59>        call   0x400550 <fopen@plt>
 →   0x400799 <main+64>        mov    QWORD PTR [rbp-0x28], rax
     0x40079d <main+68>        cmp    QWORD PTR [rbp-0x28], 0x0
     0x4007a2 <main+73>        jne    0x4007bc <main+99>
     0x4007a4 <main+75>        lea    rax, [rbp-0x20]
     0x4007a8 <main+79>        mov    rsi, rax
     0x4007ab <main+82>        mov    edi, 0x400876
─────────────────────────────────────────────────────────────── source:vsnprintf.c+20 ──
         int main ()
         {
            FILE * pFile;
            char szFileName[]="myfile.txt";

                // pFile=0x00007fffffffe538  →  0x0000000000000000, szFileName=0x00007fffffffe540  →  "myfile.txt"
 →          pFile = fopen (szFileName,"r");
            if (pFile == NULL)
              PrintFError ("Error opening '%s'",szFileName);
            else
            {
              // file successfully open
────────────────────────────────────────────────────────────────────────── threads ──
[#0] Id 1, Name: "vsnprintf", stopped, reason: SINGLE STEP
──────────────────────────────────────────────────────────────────────────── trace ──
[#0] 0x400799 → Name: main()
─────────────────────────────────────────────────────────────────────────────────────
gef➤
```

Figure 6: The output of GDB during a program execution.

program as long as it is executing. we can take advantage of another useful tool: the *Debbuger* It permits to execute an executable in a controlled environment in which we are able to follow the progress of the program[9]. By using breakpoints

[8] https://en.wikipedia.org/wiki/Strace

[9] https://en.wikipedia.org/wiki/Debugger

to stop executions, we can find out the code that is actually executed by the program, and also take a look and modify the content of the registers. Image 7 shows an example of a well known and used debugger: *GDB*. It is quite used debugger that works for different programming languages like C, C++, Pascal, Fortran, etc.... The image shows an example of its usage after hitting a breakpoint: GDB shows the content of the registers and at which line in the assembly code the execution of the program has reached. You can for example use GDB in order to create a dump of the memory that you can better analyze with want we have seen in 2.1. Note that 7 shows also to which line of the source code the assembly line corresponds. Obviously this will not be possible in most of the cases when doing Reverse Engineering, since who has made the executable want to make our job harder, so the executable is not shipped with debugging metadata attached to it.

2.3 Which is the best strategy to adopt?

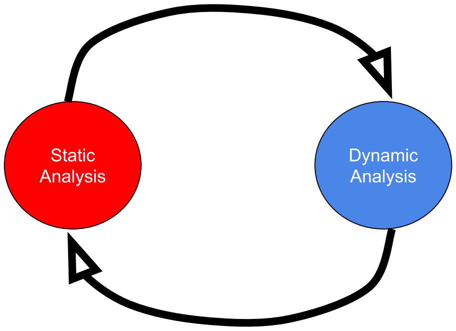


Figure 7: Relationship among Static and Dynamic Analysis.

One can wonder which approach should choose among the two we have seen so far: if the best idea is to use *Dynamic* or *Static* analysis. Both have disadvantages and advantages: for example static analysis allows to reach an higher code coverage, but if the analyzed software use some obfuscation technique it may be the case that some behaviors are shown only during the execution of the program(is the case of packer[5]), the go with Dynamic Analysis. But dynamic analysis has a low code coverage, and so some details in the behavior of a program can be missed. All of this to highlight that we have to use both approach in synergy: doing static analysis while also doing dynamic analysis, based on the situation we are dealing with. So you need to adapt you approach regarding to the software you are approaching to analyze, as we should see in the next example.

3 keycheck_baby

In this section we will cover how to apply the previous theory to a real life scenario, that in this case is the *keycheck_baby* CTF. First of all the objective of this CTF is to find the correct flag(a string) as input that is accepted as correct by the executable. In order to decide which approach we should use we can execute try to analyze the header of the file under analysis with the command as in:

```
$ file keycheck_baby
```

In our case as output you will get

```
keycheck_baby: ELF 64-bit LSB shared object, x86-64, version 1  SYSV , dynamically link
```

The important part here is that the executable is *not stripped*[10], this means that the executable contains also debug information that can be used both by a debugger but it also helps during the decompilation process. Other important information can be returned by running the following command:

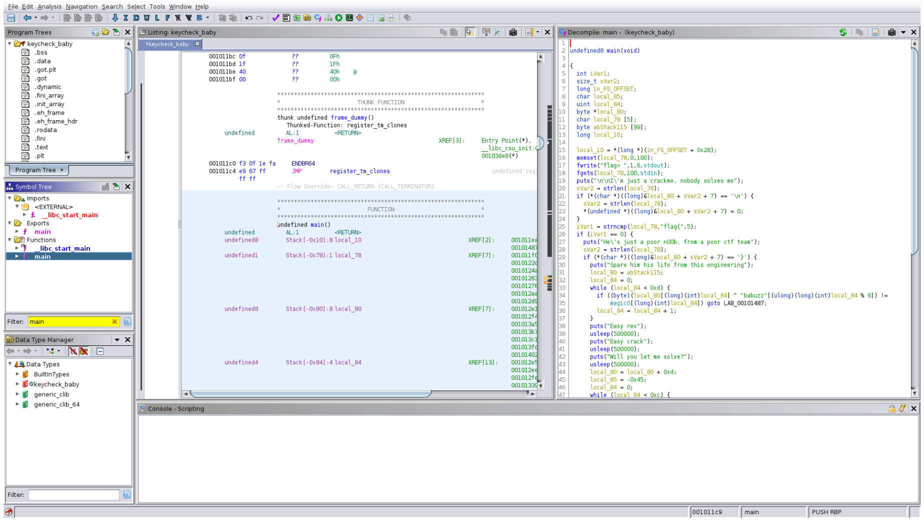
```
$ checksec keycheck_baby
```

Which gives as output:

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

Partial RELRO means that only small parts of the GOT and PLT are writable, while a canary aims in avoiding buffer overflows. NX enabled makes the stack not executable and PIE allows the code to be relocatable in case ASLR is active. The next step is to start a decompilator to try and analyze the decompiled executable. In our case we will use Ghidra. Ghidra will analyze the executable, and then we can look at the output of the disassemble and decompilation. First we need to find the entry point of the execution that can be easily found by searching inside the *Symbol Tree* by searching the *main* function, and the output will be as in 8. By taking a look a the code we can see that no obfuscation technique has been used since the whole code is there: so we can use as an approach static analysis. As we have seen in 4 in order to make things easier is a good idea to check if we can improve the code readability by retyping variable and assign a different name to them. For example by saying that the input of the *memset* function is a *char ** in our case. Now we will take a look at the most important part of the code. This first part will take as input the flag and it will store it inside the *input* variable. It checks that the string terminate with

[10] https://en.wikipedia.org/wiki/Stripped_binary



```
001011bc 0f          ??         0Fh
001011bd 1f          ??         1Fh
001011be 40          ??         40h    @
001011bf 00          ??         00h

                 **************************************************************
                 *                        THUNK FUNCTION                      *
                 **************************************************************
                 thunk undefined frame_dummy()
                   Thunked-Function: register_tm_clones
     undefined         AL:1           <RETURN>
                 frame_dummy                                  XREF[3]:     Entry Point(*),
                                                                           __libc_csu_init:(
                                                                           00103de8(*)
001011c0 f3 0f 1e fa     ENDBR64
001011c4 e9 67 ff        JMP        register_tm_clones                       undefined regi
         ff ff
                 -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

                 **************************************************************
                 *                          FUNCTION                          *
                 **************************************************************
                 undefined main()
     undefined         AL:1           <RETURN>
     undefined8        Stack[-0x10]:8 local_10                       XREF[2]:     001011ea
                                                                                  00101487
     undefined1        Stack[-0x78]:1 local_78                       XREF[7]:     001011f0
                                                                                  0010122d
                                                                                  0010124a
                                                                                  00101263
                                                                                  00101278
                                                                                  001012aa
                                                                                  001012d9
     undefined8        Stack[-0x80]:8 local_80                       XREF[7]:     001012e1
                                                                                  001012f4
                                                                                  001013a5
                                                                                  001013b7
                                                                                  001013c1
                                                                                  001013fc
                                                                                  00101402
     undefined4        Stack[-0x84]:4 local_84                       XREF[13]:    001012e5
                                                                                  001012ee
                                                                                  001012fe
                                                                                  00101339
```

```

undefined8 main(void)

{
  int iVar1;
  size_t sVar2;
  long in_FS_OFFSET;
  char local_85;
  uint local_84;
  byte *local_80;
  char local_78 [5];
  byte abStack115 [99];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  memset(local_78,0,100);
  fwrite("flag> ",1,6,stdout);
  fgets(local_78,100,stdin);
  puts("\n\nI\'m just a crackme, nobody solves me");
  sVar2 = strlen(local_78);
  if (*(char *)((long)&local_80 + sVar2 + 7) == '\n') {
    sVar2 = strlen(local_78);
    *(undefined *)((long)&local_80 + sVar2 + 7) = 0;
  }
  iVar1 = strncmp(local_78,"flag{",5);
  if (iVar1 == 0) {
    puts("He\'s just a poor n00b, from a poor ctf team");
    sVar2 = strlen(local_78);
    if (*(char *)((long)&local_80 + sVar2 + 7) == '}') {
      puts("Spare him his life from this engineering");
      local_80 = abStack115;
      local_84 = 0;
      while (local_84 < 0xd) {
        if ((byte)(local_80[(long)(int)local_84] ^ "babuzz"[(ulong)(long)(int)local_84 % 6]) !=
            magic0[(long)(int)local_84]) goto LAB_00101487;
        local_84 = local_84 + 1;
      }
      puts("Easy rev");
      usleep(500000);
      puts("Easy crack");
      usleep(500000);
      puts("Will you let me solve?");
      usleep(500000);
      local_80 = local_80 + 0xd;
      local_85 = -0x45;
      local_84 = 0;
      while (local_84 < 0xc) {
```

Figure 8: The output of GDB after analyzing the keycheck executable.

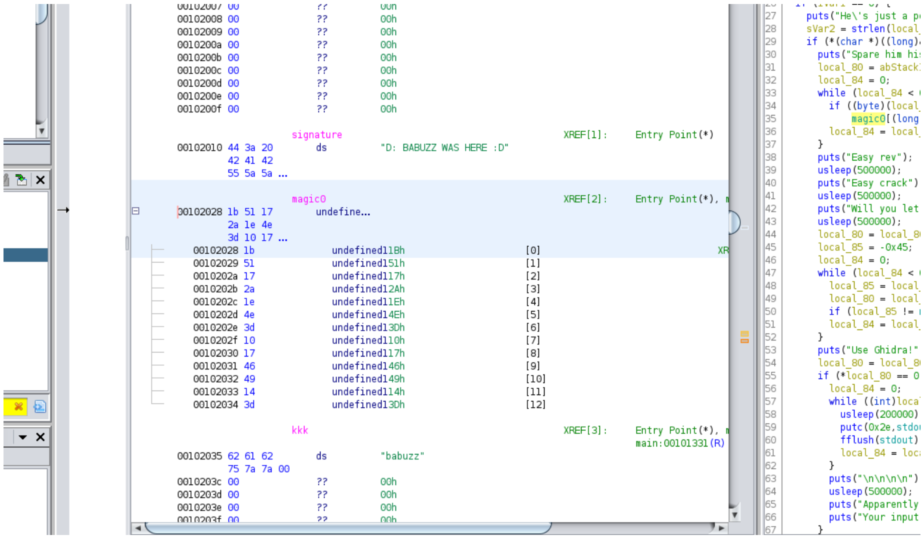
the string terminator. At the next step the input is compared with the *flag{* string:

```
iVar1 = strncmp(input,"flag{",5);
if (iVar1 == 0) {
  puts("He\'s just a poor n00b, from a poor ctf team");
  sVar2 = strlen(input);
  if (*(char *)((long)&input + sVar2) == '}') {
```

and it also verifies that the flag ends with a }. At this point the program checks the first 0xd(13) characters of the input string, by accessing it with the *index*. And it checks that the value of the input string at position index elevated at the casted int value of the character present in the *babuzz* string at position index % 6 is the same as the value contained inside the global buffer *maigc0* at position index.

```
  index = 0;
  while (index < 0xd) {
      if ((byte)(input[(long)(int)index] ^ "babuzz"[(ulong)(long  int            6
          magic0[(long)(int)index]) goto LAB_00101487;
      index = index + 1;
  }
```

We can find the content of this global buffer from Ghidra, if we double click on it we get the output as in 9. The content of this buffer is represented by the byte stored at each address on the left in the hexadecimal notation. If this is not the case the flow of execution jumps at the end of the code at the label



```
00102007 00          ??        00h
00102008 00          ??        00h
00102009 00          ??        00h
0010200a 00          ??        00h
0010200b 00          ??        00h
0010200c 00          ??        00h
0010200d 00          ??        00h
0010200e 00          ??        00h
0010200f 00          ??        00h

                   signature                                  XREF[1]:     Entry Point(*)
00102010 44 3a 20    ds        "D: BABUZZ WAS HERE :D"
         42 41 42
         55 5a 5a ...

                   magic0                                     XREF[2]:     Entry Point(*), m
00102028 1b 51 17    undefine...
         2a 1e 4e
         3d 10 17 ...
   00102028 1b          undefined11Bh                [0]                       XR
   00102029 51          undefined151h                [1]
   0010202a 17          undefined117h                [2]
   0010202b 2a          undefined12Ah                [3]
   0010202c 1e          undefined11Eh                [4]
   0010202d 4e          undefined14Eh                [5]
   0010202e 3d          undefined13Dh                [6]
   0010202f 10          undefined110h                [7]
   00102030 17          undefined117h                [8]
   00102031 46          undefined146h                [9]
   00102032 49          undefined149h                [10]
   00102033 14          undefined114h                [11]
   00102034 3d          undefined13Dh                [12]

                   kkk                                        XREF[3]:     Entry Point(*), m
                                                                           main:00101331(R)
00102035 62 61 62    ds        "babuzz"
         75 7a 7a 00
0010203c 00          ??        00h
0010203d 00          ??        00h
0010203e 00          ??        00h
0010203f 00          ??        00h
```

```
    puts("He\'s just a p
    sVar2 = strlen(local_
    if (*(char *)((long)
      puts("Spare him hi
      local_80 = abStack
      local_84 = 0;
      while (local_84 <
        if ((byte)(local_
            magic0[(long
        local_84 = local_
      }
      puts("Easy rev");
      usleep(500000);
      puts("Easy crack")
      usleep(500000);
      puts("Will you let
      usleep(500000);
      local_80 = local_8
      local_85 = -0x45;
      local_84 = 0;
      while (local_84 <
        local_85 = local_
        local_80 = local_
        if (local_85 !=
        local_84 = local_
      }
      puts("Use Ghidra!"
      local_80 = local_8
      if (*local_80 == 0
        local_84 = 0;
        while ((int)loca
          usleep(200000)
          putc(0x2e,stdo
          fflush(stdout)
          local_84 = loc
        }
        puts("\n\n\n\n")
        usleep(500000);
        puts("Apparently
        puts("Your input
      }
```

Figure 9: The output of GDB after analyzing the keycheck executable.

LAB_00101487 before checking the canary, the program exits, so it has not accepted the flag.

```
    LAB_00101487:
    if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
        return 0;
    }
```

The next check the executable does is the one in the next snippet of code: first it moves the input forward to the last tested value of it, then it uses a temp value that is initialize to a constant value: 187.

```
    input = input + 0xd;
    temp = 187;
    index = 0;
    while (index < 0xc) {
        temp = temp + *input;
        input = input + 1;
        if (temp != magic1[(long)(int)index]) goto LAB_00101487;
            index = index + 1;
    }
```

At each iteration it compares the content of this temp value and the content of another global buffer *magic1* at the value pointed by the index. Index that is incremented at each iteration until it reaches the value of *0xc*(12). The value of temp is also updated at each iteration by adding the value contained in the input string at the character that is actually pointed at this iteration, and then used for validation in the check. Pay attention that the type of temp is *char*, so it must have a value in between 0 and 255, if it goes beyond such value it is casted. If the execution is able to pass all these checks it finally prints:

```
puts("Apparently we let go (:");
puts("Your input looks like the correct flag \\(^o^)/");
```

So if we reach this part of code means that we've got the correct flag.

3.1 Exploit

Up to know we have been able in this case to perform only static analysis: with all the information we've obtained with such an analysis we can try to write an exploit that is able to compute the correct flag.

```
# Get the content of the two buffer used during the checks
magic0 =
    b'\x1b\x51\x17\x2a\x1e\x4e\x3d\x10\x17\x46\x49\x14\x3d'
magic1 =
    b'\xeb\x51\xb0\x13\x85\xb9\x1c\x87\xb8\x26\x8d\x07'
babuzz = "babuzz"

# Firs part of the flag that is checked
flag = 'flag{'

# This is the first check done by the executable
# It checks the characters in the input string from
# position len("flag{") to len("flag{")+12 against magic0.
# And its aim is to find the correct printable character
# in range between 0 and 256 that pass the checks in the
# given input position
for i in range(0,13):
    for c in range(256):
        # Add the value to the flag string if c
        # elevated at the casted value to
        # int of babuzz string at position i%6
        if int(c) ^ ord(babuzz[i%6]) == magic0[i]:
            flag = flag + chr(c)
            break

# This is the second check done by the executable
# It checks the characters in the input string from
# position len("flag{")+13 to len("flag{")+25 against
# the content of magic1.
# As before it checks the condition against all the
# printable character
# Start temp with a constant value
const = 187
for i in range(0,12):
    for c in range(256):
        # Update the value stored in temp, and cast
        # it to be a char so it needs to have a value
        # between 0 and 256, thus the % operation
        temp = (const + c) % 256
        if temp == magic1[i]:
            const = temp
            flag = flag + chr(c)
            break

# Add to the obtained flag the closing character
flag = flag + "}"

# Finally print the obtained flag
print(flag)
```

This python script has been written by taking advantage of the information we have got during static analysis. It has been written by trying to emulate the behavior of the executable reconstructed after disassembling it. It assumes that the characters present inside the flag are printable ASCI character, so their int value goes from 0 to 255. The python script replicate the checks done by the executable and tries to find among all the printable characters the ones that are able to pass such checks. So if we run the script we get as output:

```
$ flag y0u_d4_qu33n_0f_cr4ck1ngz
```

That is the flag accepted by the executable if provided to it as input.

4 Conclusion

This document provides an introduction to Reverse Engineering applied to the field of Software Engineering. It illustrates different techniques and approaches used in order to reverse engineering an executable from scratch, by using different tools: it then applies all of this to a real example taken from a CTF, in which applying the previously introduced tools, providing also a working exploit for it.

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.

[5] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1), July 2013.

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