

# How to solve "Introduction to Reverse Engineering 1"

---

Reverse engineering is an art form. The art of understanding a mechanism step by step in order to finally understand it so well that it can be exploited or even bypassed. In concrete terms, reverse engineering is about analyzing binary executables. These can be programs of most different programming languages on most different operating systems. Especially in the subject area of reverse engineering, one learns to be flexible and to quickly grasp new contexts.

Often one has to do in CTFs with compiled Unix programs. Therefore for this introduction task also an ELF file was created, which is to be analyzed with most different approaches. In the following sections different techniques and programs are presented, which make such an analysis possible. Not all methods lead to the goal, but each method reveals something about the program to be analyzed.

The structure of a binary Linux program would go beyond the scope of this introduction. It should be said, however, that the program is in the form of machine instructions. These are understood by the processor and processed sequentially. The beginning of each program is called an `EntryPoint`, usually in the function `main`.

The programs and methods discussed now are only touched upon and actually deserve their own, multi-page explanations. But with this basic information it is possible to learn more about the tools and methods and to experiment by yourself!

## Get an overview of the program

---

At the beginning it is useful to get to know the program to be analyzed. The easiest way to do this is to run it and play around with it to get to know its functionality:

```
$ ./rev1
Give me your password:
I_DONT_KNOW
Thats not the password!
```

The program seems to ask for a password and quits as soon as it encounters an incorrect password entry, in this case `I_DONT_KNOW`. So we have to find out the password hidden in the program to get the desired flag.

The determined password has to be entered later at the server of the CSCG platform to get the final flag.

# Static methods

---

Static methods are suitable for the thorough, structured analysis of binary programs. This is done mainly in the form of disassemblers and decompilers. Disassemblers turn binary machine instructions into readable assembly code. This lowest form of any programming language reduces the program to arithmetic operations, comparisons and jumps within the program. The machine instructions are thereby in the CPU architecture in which the program is later executed. Usually this is `x86_64`, but it can also be `ARM`, `MIPS` or `RISC-V`.

## file

If you have no idea what kind of file it might be, `file` often helps. This little program recognizes many different file formats and spits out information about them. In a CTF some time ago `file` gave the decisive hint by recognizing a `MS DOS Bootloader` in a binary file.

A call to `file` is done via the console:

```
file rev1
rev1: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, inte
```

This output tells us that it is an `x86_64`, i.e. 64 bit, ELF program in the x86 architecture whose metadata has not been stripped ( `non stripped` ).

## objdump

`objdump` is a program that is often pre-installed in Linux environments. Among other things, it is suitable for examining ELF files and disassembling the machine instructions they contain. The machine instructions are obtained by calling `objdump -M intel -d rev1`.

One is often overwhelmed by the many instructions, so `objdump` is usually only suitable as a rough aid for basic orientation. One notices here very fast that it makes sense to use a real disassembler.

## Ghidra

The entire IT security scene was excited when this tool, developed by the NSA, was released for free use a few years ago. Ghidra can open a wide variety of architectures and file formats and combine the metadata present there in a meaningful way.

In addition to a clear disassembly display, Ghidra attempts to reconstruct the C code that led to the generation of the machine instructions. In more demanding tasks, however, the reconstructed C code is often erroneous and more confusing than the individual instructions of the program. Therefore, one should never rely on the reliability of this reconstructed code.

After loading and initial analysis of the program by Ghidra, the ELF headers are displayed. These are not relevant for the task at hand. By pressing `g` (go to) and entering `main` it is possible to jump directly to the entry point. In this function the actual program starts after it has been loaded into memory by the so-called `loader`. In the disassembly one identifies different calls and conditions:

```
CALL initialize_flag => Reads the flag from the current directory
[...]
CALL puts => Console output of "Give me your password".
[...]
CALL read => Read console input
[...]
CALL strcmp => Compare two strings, return to EAX
TEST EAX,EAX => If EAX == 0
JNZ LAB_0010093a => Jump, if the last result was not correct
CALL puts => Output of "Thats the right password!"
[...]
```

The C functions like `puts`, `read` and `strcmp` can be found in [References](#). For this case the function `strcmp` is of interest. You can see from the documentation that this function returns `0` if the two strings to be compared are exactly the same. From this one can draw the conclusion that probably a password is checked there. If the two strings are identical, `Thats the right password!` is returned and the flag is printed.

The password itself can be read directly in the C representation, the so-called decompile. Before the `strcmp` function is called, the memory pointing to the password but also some instructions before the `strcmp` is loaded into the register `RDI`. This corresponds to the first argument in the `x86_64` calling convention and thus the first parameter of the `strcmp` function.

Finding this string is left to the reader.

## Strings

Strings are representations of ASCII characters, i.e. numerical values that lie in a specific range. For example, an `A` corresponds to the number `0x41 = 65` in the decimal system. With time, one knows the ASCII range and is able to recognize ASCII in numerical values. Until then, however, an [ASCII table](#) helps. The program 'strings' tries to read as many ASCII values as possible from binary data. So if many numeric values fall one after the other into the ASCII range that can be displayed, they are recognized as strings and output.

This small utility is usually pre-installed and can be called via `strings rev1`:

```
Give me your password:
*****

Give me your password:
***** [REDACTED PASSWORD] *****

Thats the right password!
```

```

Flag: %s
Thats not the password!
./flag.txt
flag.txt
File "%s" not found. If this happens on remote, report to an admin. Otherwise, please
;*3$"
GCC: (Debian 8.3.0-6) 8.3.0

[...]
strcmp@@GLIBC_2.2.5
[...]
main
fopen@@GLIBC_2.2.5
initialize_flag

```

Strings finds a lot of information in the program to be analyzed, e.g. the used compiler `gcc 7.4.0`, the already known console outputs and functions registered in the program like `strcmp` and `initialize_flag`. Since the password to be guessed can also be found via `strings` and again left to the reader to practice. In the above output the password has been replaced by the string `*****`

[REDACTED PASSWORD] \*\*\*\*\* .

## Dynamic methods

Programs often calculate elaborate checksums or decrypt their program code at runtime. These tricks, which should make life difficult for a reverse engineer, are difficult to capture using static methods. It would therefore help to be able to observe individual functions and their parameters and return values at runtime.

### ltrace and strace

A trace program observes the program flow and stores the called functions during this time. In doing so, the tracer usually hangs between the program to be executed and the operating system. `ltrace` observes the `usercalls`, i.e. the already known C-functions like `puts` and `strcmp`, while `strace` records the `syscalls`. Syscalls are direct interactions with the operating system, e.g. to do a console output.

For the program at hand, `usercalls` and thus `ltrace` are more interesting. An execution of `ltrace ./rev1` intersects all known C functions and prints their call, parameter and return value on the console. In addition to the already known strings of the console output, one also sees the comparison between the user input and the password to be found:

```

fopen("./flag", "r")
fread(0x5598546e2040, 256, 1, 0x559855172260)
fclose(0x559855172260)
puts("Give me your password: "Give me your password:
)

```

= 24

```

read(0I_DONT_KNOW
, "I_DONT_KNOW\n", 31)
strcmp("I_DONT_KNOW", "*****")
puts("Thats not the password!"Thats not the password!
)
+++ exited (status 0) +++
= 24

```

First the `flag` file is read via `fopen` and stored in memory. Then `puts` is called, followed by a `read` to read the console input. After that, the comparison between the secret password and the user input takes place, which returns a return value not equal to `0`. This causes the error message to be issued. Deciding the correct password via `ltrace` is again left to the reader.

## **gdb**

Probably the best known debugger under Unix is `gdb`, the abbreviation for GNU Debugger. This console based debugger is very powerful, but without extensions it is unintuitive to use, especially in the beginning. Therefore I recommend to use a GDB extension like `pwndbg` or `gef`. These plugins have advantages and disadvantages, but each one allows a much better handling of `gdb`. In this example, `pwndbg` is used.

The debugger can be started with the command `gdb ./rev1`. To run the program, it is sufficient to type `run` in the debugger console. Since no further settings were made, the program follows its program flow until it is terminated.

```

pwndbg> run
Starting program: /home/theuser/Downloads/cscg20/challenges/intro_rev/deploy/rev1/rev1
Give me your password:
I_DONT_KNOW
Thats not the password!
[Inferior 1 (process 12333) exited normally]
pwndbg>

```

It gets interesting when we follow the program step by step. For this purpose it is sufficient to set a breakpoint at the entry point `main`:

```

pwndbg> br main
Breakpoint 1 at 0x11a9

```

The debugger stops the program as soon as a function with a breakpoint is reached. From now on the values of the registers can be inspected in peace and the execution can be continued in single machine instructions. With `n` the next instruction is executed. Before each function call `pwndbg` displays the passed parameters, if the function is known. For example, the first `puts` call after `n` has been entered seven times:

```

0x555555551a9 <main+4>    sub    rsp, 0x30
0x555555551ad <main+8>    mov    eax, 0
0x555555551b2 <main+13>   call   initialize_flag          <initialize_flag>

0x555555551b7 <main+18>   lea    rdi, [rip + 0xe4a]
► 0x555555551be <main+25>   call   puts@plt                <puts@plt>
    s: 0x555555556008 ← 'Give me your password: '

0x555555551c3 <main+30>   lea    rax, [rbp - 0x30]
0x555555551c7 <main+34>   mov    edx, 0x1f
0x555555551cc <main+39>   mov    rsi, rax
0x555555551cf <main+42>   mov    edi, 0
0x555555551d4 <main+47>   call   read@plt                <read@plt>

```



Using this technique we can reach the `strcmp` function:

pwndbg>

0x0000555555551f7 in main ()

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS

```

RAX 0x7fffffffef4c0 ← 'I_DONT_KNOW'
RBX 0x0
RCX 0x7ffff7ece311 (read+17) ← cmp rax, -0x1000 /* 'H=' */
RDX 0x1f
*RDI 0x7fffffffef4c0 ← 'I_DONT_KNOW'
RSI 0x555555556020 ← 'm4gic_passw0rd'
R8 0x3
R9 0x77
R10 0x0
R11 0x246
R12 0x555555550c0 (_start) ← xor ebp, ebp
R13 0x7fffffffef5d0 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffef4f0 → 0x555555552b0 (__libc_csu_init) ← push r15
RSP 0x7fffffffef4c0 ← 'I_DONT_KNOW'
*RIP 0x555555551f7 (main+82) ← call 0x555555555080

```

[ D

```

0x555555551e2 <main+61>    cdqe
0x555555551e4 <main+63>    mov    byte ptr [rbp + rax - 0x30], 0
0x555555551e9 <main+68>    lea    rax, [rbp - 0x30]
0x555555551ed <main+72>    lea    rsi, [rip + 0xe2c]
0x555555551f4 <main+79>    mov    rdi, rax
► 0x555555551f7 <main+82>    call   strcmp@plt            <strcmp@plt>
    s1: 0x7fffffffef4c0 ← 'I_DONT_KNOW'
    s2: 0x555555556020 ← '*****'

0x555555551fc <main+87>    test   eax, eax

```

Both in the displayed parameters, but also in the corresponding registers 'RDI' and 'RSI' the searched password can be read out.

## Conclusion

---

As shown in the last sections, in this very simple task the password can be found in several ways. Each of the tools takes a different approach and allows meaningful insights to be gained from the binary 1s and 0s.

Heavy reverse engineering tasks take active countermeasures to prevent such techniques: strings are often stored "encrypted" so that they cannot be recognized as ASCII. There are programming techniques that terminate the program as soon as an active debugger is detected. Thus a cat-and-mouse game develops between the Reverser and the programmer, who wants to prevent the Reverse engineering. In the next two tasks, such initial measures have been taken to prevent the easy reading of the password using at least some of the techniques shown here. Have fun!