Linux ELF Files & Binary Analysis

Emre Kosar

Table of Contents

Contact Me

Thank you for reviewing my documentation. If you have any questions or see any incorrect or missing information, please do not hesitate to share with me. Here is my LinkedIn page in order to contact.

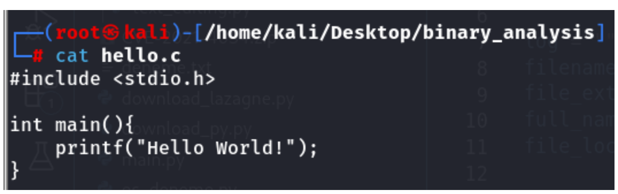
https://www.linkedin.com/in/emre-ko%C5%9Far-68087a252/

What is Binary File?

A binary file is a file whose content is in a binary format consisting of a series of sequential bytes, each of which is eight bits in length. The content of the binary must be interpreted by a compiler or processor that exactly understands how that content is formatted and how to read the data.

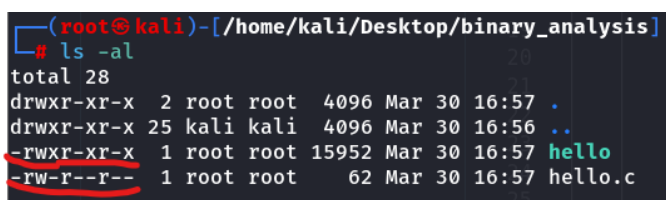
ELF Files

ELF file refers to file format which is in use in Unix-Linux systems just like **PE (Portable Executable)** file format in Windows. The filename of PE binaries ends with **.exe**, while ELF binaries do not have an extension. ELF is the abbreviation for **Executable and Linkable Format**. It defines the structure for binaries, libraries, and core files. Generally, ELF files are the output of a compiler or linker, and they are a binary format. In Linux, to determine the type of file we use '***file*** ***<filename>***' command. Here is a quick demonstration.



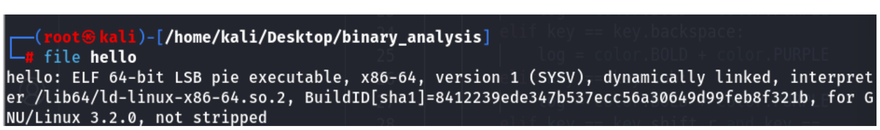
```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# cat hello.c
#include <stdio.h>

int main(){
    printf("Hello World!");
}
```

- As an example, we've created a simple script as it's seen on the image, and compiled it by using '***gcc hello.c -o hello***' command. (Output file refers '*hello*')



```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# ls -al
total 28
drwxr-xr-x  2 root root  4096 Mar 30 16:57 .
drwxr-xr-x 25 kali kali  4096 Mar 30 16:56 ..
-rwxr-xr-x  1 root root 15952 Mar 30 16:57 hello
-rw-r--r--  1 root root    62 Mar 30 16:57 hello.c
```

- As the image proves, the compiled file is also executable compared to source code file which is ***hello.c*** file.



```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=8412239ede347b537ecc56a30649d99feb8f321b, for GNU/Linux 3.2.0, not stripped
```

- To jump into the point, as seen on the image, the compiled file is an ELF file as well as compiled via 64-bit. Since it's an executable, it can be operated by putting '**./**' before the file name.

- What if the provided file was a file with '***.txt***' extension?

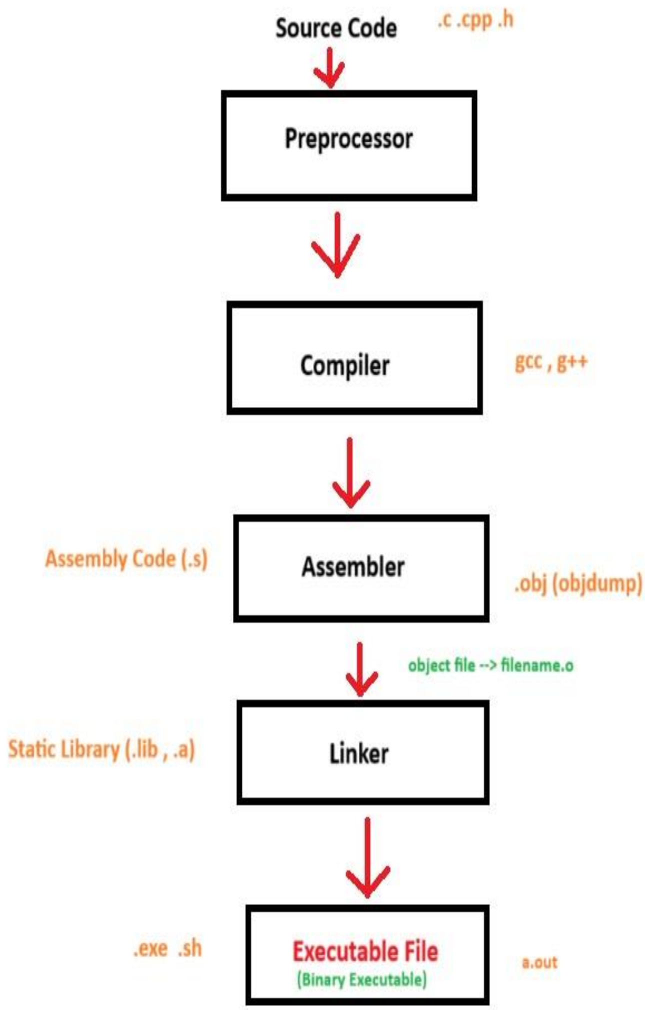
To answer this question, let's take a look at a .txt file which contains only '***hello world***' text in it.

It says that the provided text file includes ASCII text.

- Keep in mind that ASCII texts don't have any instructions which means it **cannot be used** in Binary Analysis. In other words, they cannot be disassembled by tools such as **objdump**.

Compilation Process of C File

Every ‘***.c***’ , ‘***.cpp***’ or ‘***.h***’ files need to be compiled to become executable files. The process of compiling a Binary File to executable format follows a path as you can see on the flowchart below. Let’s cover every one of the steps. But first, let’s look at the flowchart. As we can see at the top of the flowchart, we need a source code file.



Preprocessing Step in C File Compiling

First of all, preprocessors are programs that process the source code before compilation. Preprocessor directives begin with # symbol. The '#' symbol indicates that whatever statement starts with a '#' will go to the preprocessor program to get executed.

Mainly, there are four types of Preprocessor Directives. Let's clarify them one by one.

```
#include <stdio.h>

#define isim "Emre"

int main(){
    printf(isim);
}
```

1-Macros: They are pieces of code in a program that have a token-value relationship. Based on the sample script above, output will be the corresponding value of '***isim***', as defined as '***Emre***'.

2-File Inclusion: This type of preprocessor directive tells the compiler to include a file in the source code program. As its name suggests, '***#include***' is used to include the *header files* in the C program. Header files could be both standard header files or user-defined header files (For standard headers; ***#include <file_name.h>*** , For user-defined headers; ***#include filename***).

3-Conditional Compilation: This is a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions. (***#ifdef*** , ***#ifndef*** , ***#if*** , ***#elif*** , ***#else*** , ***#endif***)

4-Other Directives: Apart from other directives mentioned, there are two more and they are not commonly used. (***#undef*** and ***#pragma***)

- When we'd like to not compile, but only preprocess, we are going to need to use **-E** parameter of gcc.

```
-E                            Preprocess only; do not compile, assemble or link.
```

Converting Source Code To Assembly In Assembling Process

The point we will be talking about in this section is to create ***filename.s*** by using ***Assembler method***. This section consists of the '***Assembler***' step in diagram of Compilation of C file above. We will assume the source code as the same script provided above. Yet, it will also be provided below.

- To convert source code to Assembly, we're going to need to use ***-S*** parameter of *gcc*.

```
-S                            Compile only; do not assemble or link.
```

As the description of '***-S***' parameter enlightens us, this parameter doesn't assemble or link, only for compiling. Here is how to use:

```
──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# gcc -S file.c
```

```
-rw-r--r--  1 root root    73 Apr  3 17:44 file.c
-rw-r--r--  1 root root   475 Apr  4 06:18 file.s
```

The '***-S***' parameter produced a new file with ***.s*** extension. As mentioned above, since '***-S***' parameter doesn't assemble or link, output file is not an executable file.

Let's examine the '***file.s***' file.

home > kali > Desktop > binary_analysis > file.s

```
	.file	"file.c"
	.text
	.section	.rodata
.LC0:
	.string "Emre"
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	leaq	.LC0(%rip), %rax
	movq	%rax, %rdi
	movl	$0, %eax
	call	printf@PLT
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
	.ident	"GCC: (Debian 13.2.0-7) 13.2.0"
	.section	.note.GNU-stack,"",@progbits
```

➢ As we can see, the source code (.c) has turned into Assembly file. Let's break it down. First, ***.file*** section specifies the file which was generated from.

.rodata section stands for **Read-Only Data**.

.LC0 section is the location. Under this location, we can see the defined string value, and main function.

Directives which start with '***.cfi***' result in generation of additional data by the compiler.

'***.LF***' section stands for '***leave* and *ret***'. They are CPU instructions.

Creating Binary (ELF) File from Object File

An object file is the real output from the compilation phase. It's most likely machine code (binary code) but has information that allows a linker to see what symbols it requires in order to execute. (Symbols represent names of global objects such as functions.)

In this step, we won't be linking the assembly file created on previous section, only compile, and assemble. In order to make it happen, we will use **-c** parameter. Here is the description of this parameter:

```
-c                            Compile and assemble, but do not link.
```

To use this parameter, we have to provide the Assembly file created previously, which will be the file with ***.s*** extension. Here is how to do for this case:

```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# gcc -c file.s
```

After compiling the '***file.s***' file, we have '***file.o***' as output of '***gcc -c file.s***' command.

```
-rw-r--r--  1 root root   73 Apr  3 17:44 file.c
-rw-r--r--  1 root root 1368 Apr  9 15:47 file.o
-rw-r--r--  1 root root  475 Apr  4 06:18 file.s
```

Let's take a look at what's inside of the ***file.o*** file.

```
file.o
7f45 4c46 0201 0100 0000 0000 0000 0000
0100 3e00 0100 0000 0000 0000 0000 0000
0000 0000 0000 0000 1802 0000 0000 0000
0000 0000 4000 0000 0000 4000 0d00 0c00
5548 89e5 488d 0500 0000 0048 89c7 b800
0000 00e8 0000 0000 b800 0000 005d c345
6d72 6500 0047 4343 3a20 2844 6562 6961
6e20 3133 2e32 2e30 2d37 2920 3133 2e32
2e30 0000 0000 0000 1400 0000 0000 0000
017a 5200 0178 1001 1b0c 0708 9001 0000
1c00 0000 1c00 0000 0000 0000 1f00 0000
0041 0e10 8602 430d 065a 0c07 0800 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0100 0000 0400 f1ff
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0300 0100 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0300 0500
0000 0000 0000 0000 0000 0000 0000 0000
0800 0000 1200 0100 0000 0000 0000 0000
1f00 0000 0000 0000 0d00 0000 1000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0066 696c 652e 6300 6d61 696e 0070 7269
6e74 6600 0000 0000 0700 0000 0000 0000
0200 0000 0300 0000 fcff ffff ffff ffff
1400 0000 0000 0000 0400 0000 0500 0000
fcff ffff ffff ffff 2000 0000 0000 0000
0200 0000 0200 0000 0000 0000 0000 0000
002e 7379 6d74 6162 002e 7374 7274 6162
002e 7368 7374 7274 6162 002e 7265 6c61
2e74 6578 7400 2e64 6174 6100 2e62 7373
002e 726f 6461 7461 002e 636f 6d6d 656e
7400 2e6e 6f74 652e 474e 552d 7374 6163
6b00 2e72 656c 612e 6568 5f66 7261 6d65
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 2000 0000 0100 0000
0600 0000 0000 0000 0000 0000 0000 0000
4000 0000 0000 0000 1f00 0000 0000 0000
Line 1, Column 1
```

➢ This file contains full of machine code (Binary code). If there was an error in the source code (.c file), machine code would have error either and this would indicate that CPU **cannot read the instructions**.

- As a reminder, machine code and binary code are the same. However, machine code can also be expressed in hexadecimal format, a number system with BASE16.

Linking the Object File

As the last step to get ***a.out*** based on the flowchart of compilation process, this step covers linking the object file in order to run it and get the output. For the mentioned point to work, we will be using **-o** parameter.

```
-o <file>                Place the output into <file>.
```

When using '***-o***' parameter, we will be using the file which ends with ***.o*** extension. Here is how to link the object file for this case:

```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# gcc file.o -o file
```

The command ***gcc file.o -o file*** has created a linked file named ***file*** which is an ELF file and executable. We can check it by running '***ls -l***' command to see if it's executable or not.

```
-rwxr-xr-x  1 root root 15952 Apr 12 04:21 file
-rw-r--r--  1 root root    73 Apr  3 17:44 file.c
-rw-r--r--  1 root root  1368 Apr  9 15:47 file.o
-rw-r--r--  1 root root   475 Apr  4 06:18 file.s
```

Another way to check if the '***file***' is ELF or not, is to use file command.

```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# file file
file: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=36050a6715fb3c23f69a
fa11fda2f44b02861e8f, for GNU/Linux 3.2.0, not stripped
```

Let’s run our linked object file and see what we get.

```
┌──(root㉿kali)-[/home/kali/Desktop/binary_analysis]
└─# ./file
Emre
```

As you can see, we got ‘***Emre***’ printed in the output.

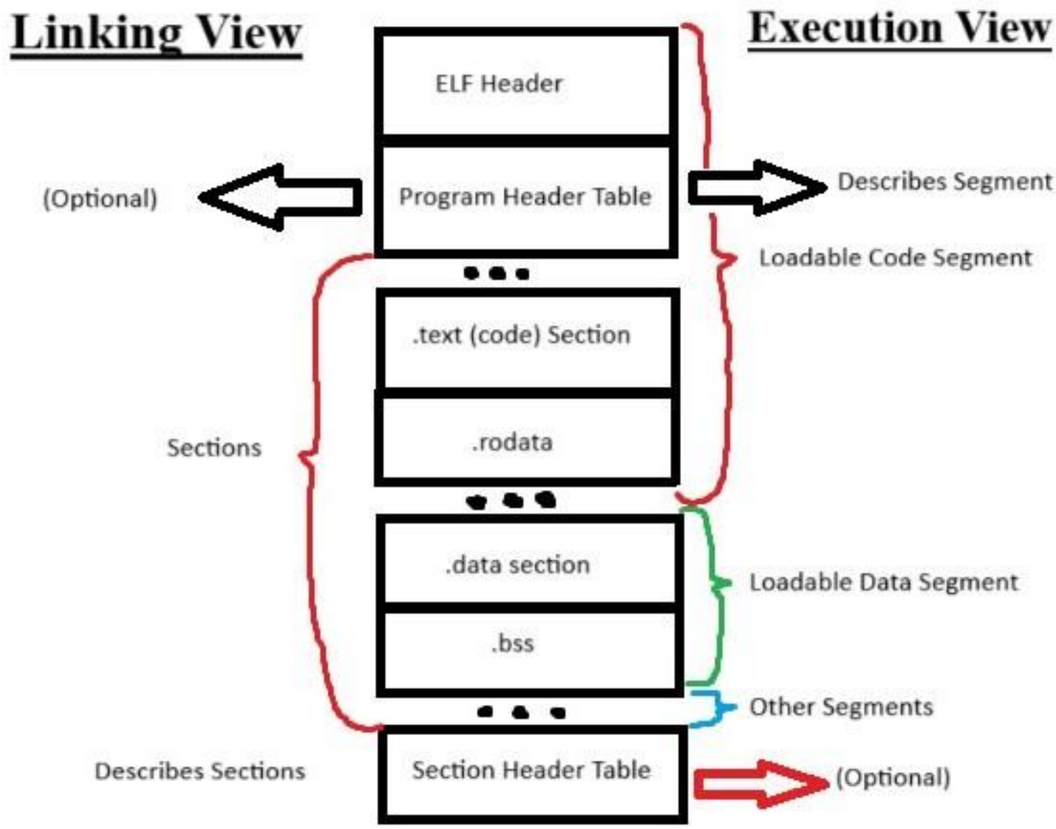
Examining Sections of ELF Files

ELF is short for ‘**Executable and Linkable Format**’. It’s a format used for storing binaries, libraries and core dumps on disks in Linux/Unix based systems. Moreover, the ELF format is versatile. Its design allows it to be executed on various processor types. This is a significant reason why the format is common compared to other executable file formats.

Generally speaking, we write most programs in High-Level languages such as C/C++, Python, Java etc. These programs cannot be directly executed on the CPU because the CPU doesn’t understand these instructions. Instead, we use a compiler that compiles the high-level language into object code. Using a linker, we also link the object code with shared libraries to get a binary file.

The Structure of the ELF File

In general, the ELF File consists of two parts. The first part is the **ELF Header**, and the second part is the **File Data**. If we go in depth, the File Data is made up of the **Program Header Table**, **Section Header Table** and **Data**. Particularly, the ELF Header is always available in the ELF File, while the Section Header Table is important during link time to create an executable. On the other hand, the Program Header Table is useful during runtime to help load the executable into memory. Let’s take a look at the ELF file structure on flowchart below.



As you can see, the ELF file consists of different sections. Let's look at these sections of ELF file in more detail.

ELF Header

First of all, the ELF Header is found at the start of the file. It starts with a sequence of four unique bytes that are **0x7F** followed by **0x45**, **0x4c**, and **0x46** which translates into the three letters E, L and F. ELF Header includes information about whether the ELF file is 32-bit or 64-bit, whether it's using little-endian or big-endian, the ELF version, and the architecture that the file requires. Particularly, the metadata in ELF header helps different processors architectures to interpret the ELF file. We are going to cover fields in ELF Header further.

- In order to see ELF Header information of an ELF file, we use ***readelf*** command and also **-h** parameter. In this case, we will be reading the binary file of '*pwd*' command.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf -h /usr/bin/pwd
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x2970
  Start of program headers:          64 (bytes into file)
  Start of section headers:          41968 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

As you can see, we have displayed the ELF file header. Let's have a closer look at the fields in the ELF header.

Magic: These are the first bytes in the ELF Header. They identify the file as an ELF and contain information that processors can use to interpret the file.

Class: The value in the class field indicates the architecture of the file. As such the ELF can either be 32-bit or 64-bit.

Data: This field specifies the data encoding. This is important to help processors interpret incoming instructions. The most common data encodings are little-endian and big-endian.

Version: Identifies the ELF file version. It's set to 1.

OS/ABI: ABI is acronym for **Application Binary Interface**. In this case, it defines how functions and data structures can be accessed in the program.

ABI Version: As its name suggests, it specifies the ABI version.

Type: The value in this field specifies the object file type. For instance, 2 is for an executable, 3 is for a shared object, and 4 is for a core file.

Machine: This specifies the architecture needed for the file.

Version: Identifies the object file version.

Entry Point Address: This indicates the address where the program should start executing. In the case that the file is not an executable file, the value in this field is set to 0.

Start of Program Headers: This is the offset on the file where the program headers start.

Start of Section Headers: This is the offset that indicates where the section headers start.

Flags: This contains flags for the file.

Size of This Header: This specifies how big the ELF Header is.

Size of Program Header: The value in this field specifies how big an individual program header is.

Number of Program Headers: This indicates how many program headers there are.

Size of Section Headers: The value in this field shows how big an individual section header is.

Number of Section Headers: This indicates how many section headers there are.

Section Header String Table Index: The section table index of the entry representing the section name string table.

Program Header Table

Another section is the Program Header Table. The program header table stores information about segments. Each segment is made up of one or more sections. The kernel uses this information at run time. It tells the kernel how to create the process and map the segments into memory.

To run a program, the kernel loads the ELF Header and the program header table into memory. Then, it loads the contents that are specified in *LOAD* in the Program Header Table into memory, and it also checks if the interpreter is needed. Finally, the control is given to the executable itself or the interpreter if it's available.

- In order to see Program Header Table information of an ELF file, we use ***readelf*** command again with ***-l*** parameter. In this case, we will be reading the binary file of '*pwd*' command as we did in ELF Header section. It will be on the next page to see it clearly.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf -l /usr/bin/pwd

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x2970
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000001680 0x0000000000001680  R      0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x0000000000004729 0x0000000000004729  R E    0x1000
  LOAD           0x0000000000007000 0x0000000000007000 0x0000000000007000
                 0x0000000000001e90 0x0000000000001e90  R      0x1000
  LOAD           0x0000000000009cd0 0x0000000000009cd0 0x0000000000009cd0
                 0x0000000000000570 0x0000000000000710  RW     0x1000
  DYNAMIC        0x0000000000009dd8 0x0000000000009dd8 0x0000000000009dd8
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000000358 0x0000000000000358
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x0000000000007cb0 0x0000000000007cb0 0x0000000000007cb0
                 0x0000000000000324 0x0000000000000324  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000009cd0 0x0000000000009cd0 0x0000000000009cd0
                 0x0000000000000330 0x0000000000000330  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .data.rel.ro .dynamic .got .got.plt .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
   10     .eh_frame_hdr
   11
   12     .init_array .fini_array .data.rel.ro .dynamic .got
```

- As you can see, we have displayed the Program Header Table. Program Headers are essential when running the executable because they tell the OS all it needs to know to put the executable into memory and run it.

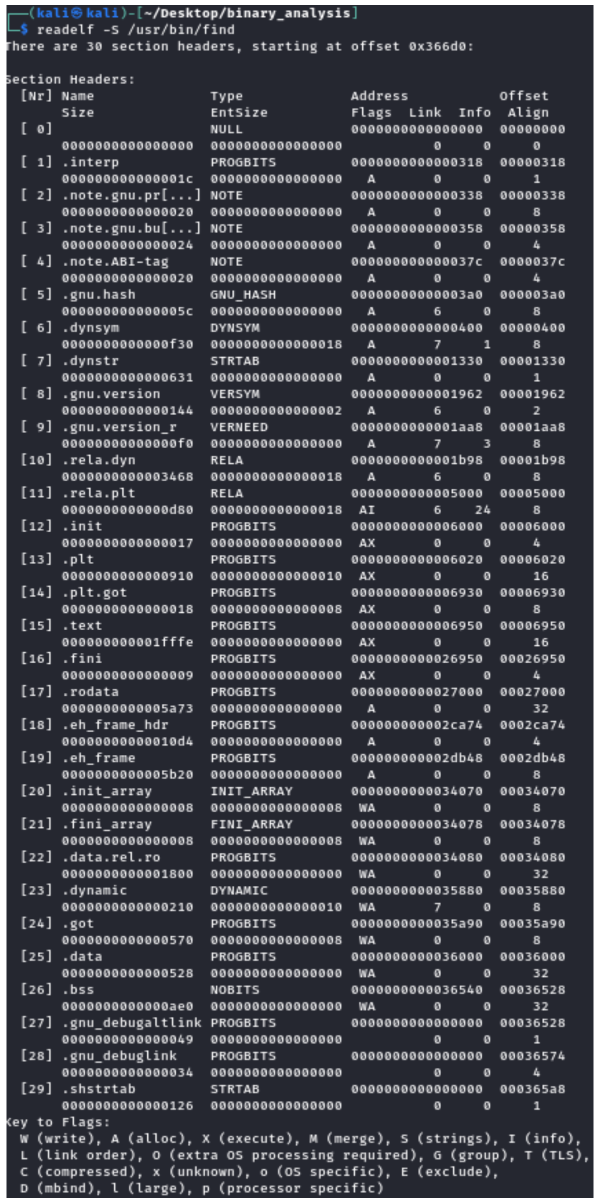
Section Header Table

As its name suggests, the Section Header Table stores and isolates data about sections. The data inside the Section Header Table is used before the execution of program, in other words during the dynamic link time.

A linker links the binary file with shared libraries that it needs by loading them into memory. The linker's implementation is specific to the OS.

Additionally, the Section Header Table contains data that's used by other files to find the symbolic definitions and references of the program.

- In order to see Section Header Table information of an ELF File, we use '***readelf***' command with **-S** parameter. In this case, we will be reading the binary file of '*find*' command.



```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf -S /usr/bin/find
There are 30 section headers, starting at offset 0x366d0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000358  00000358
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003a0  000003a0
       000000000000005c  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           0000000000000400  00000400
       0000000000000f30  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000001330  00001330
       0000000000000631  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           0000000000001962  00001962
       0000000000000144  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000001aa8  00001aa8
       00000000000000f0  0000000000000000   A       7     3     8
  [10] .rela.dyn         RELA             0000000000001b98  00001b98
       0000000000003468  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000005000  00005000
       0000000000000d80  0000000000000018  AI       6    24     8
  [12] .init             PROGBITS         0000000000006000  00006000
       0000000000000017  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000006020  00006020
       0000000000000910  0000000000000010  AX       0     0     16
  [14] .plt.got          PROGBITS         0000000000006930  00006930
       0000000000000018  0000000000000008  AX       0     0     8
  [15] .text             PROGBITS         0000000000006950  00006950
       000000000001fffe  0000000000000000  AX       0     0     16
  [16] .fini             PROGBITS         0000000000026950  00026950
       0000000000000009  0000000000000000  AX       0     0     4
  [17] .rodata           PROGBITS         0000000000027000  00027000
       0000000000005a73  0000000000000000   A       0     0     32
  [18] .eh_frame_hdr     PROGBITS         000000000002ca74  0002ca74
       00000000000010d4  0000000000000000   A       0     0     4
  [19] .eh_frame         PROGBITS         000000000002db48  0002db48
       0000000000005b20  0000000000000000   A       0     0     8
  [20] .init_array       INIT_ARRAY       0000000000034070  00034070
       0000000000000008  0000000000000008  WA       0     0     8
  [21] .fini_array       FINI_ARRAY       0000000000034078  00034078
       0000000000000008  0000000000000008  WA       0     0     8
  [22] .data.rel.ro      PROGBITS         0000000000034080  00034080
       0000000000001800  0000000000000000  WA       0     0     32
  [23] .dynamic          DYNAMIC          0000000000035880  00035880
       0000000000000210  0000000000000010  WA       7     0     8
  [24] .got              PROGBITS         0000000000035a90  00035a90
       0000000000000570  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000036000  00036000
       0000000000000528  0000000000000000  WA       0     0     32
  [26] .bss              NOBITS           0000000000036540  00036528
       0000000000000ae0  0000000000000000  WA       0     0     32
  [27] .gnu_debugaltlink PROGBITS         0000000000000000  00036528
       0000000000000049  0000000000000000           0     0     1
  [28] .gnu_debuglink    PROGBITS         0000000000000000  00036574
       0000000000000034  0000000000000000           0     0     4
  [29] .shstrtab         STRTAB           0000000000000000  000365a8
       0000000000000126  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

- As we can see, the index number [0] contains NULL. Since it's NULL, it's not reflected in string table.

'.init' and '.fini' Sections in ELF File

The name of **.init** section comes from the word *initialization*. The '**.init**' section contains the instructions for when the program is started. These instructions usually refer to dynamic memory allocation, initializing global variables, or other initialization processes in order to start the program.

The name of **.fini** section comes from the word *finalization*. The '**.fini**' section contains the instructions to contribute to the termination process of the program.

The '**.init**' and '**.fini**' sections have a special purpose. If a function is placed in the '**.init**' section, the system will execute it before the main function. Additionally, the functions placed in the '**.fini**' section will be executed by the system after the main function returns. This feature is utilized by compilers to implement global constructors and destructors.

When ELF executable is executed, the system will load in all the shared object files before transferring control to the executable. With the properly constructed '**.init**' and '**.fini**' sections, constructors and destructors will be called in the right order.

Let's take a look at the '**.init**' and '**.fini**' functions on the example.

```c
#include <stdio.h>


void __attribute__((constructor)) init_func(void){
    printf(".init function is called.\n\n");
}


void __attribute__((destructor)) fini_func(void){
    printf(".fini function is called.\n\n");
}

int main(void){
    printf("Main function is called.\n\n");
    return 0;
}
```

- This script should run 'init_func', 'main' and 'fini_func' functions in this order. Let's compile and run it and see how this script works.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ gcc -o fini_init fini_init.c

┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ ./fini_init
.init function is called.

Main function is called.

.fini function is called.
```

➢ As we can see, the program has worked the way that it should be.

Let’s dissemble the program and take a look at the ‘.init’ and ‘.fini’ sections by using *objdump*.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf --wide -S fini_init
There are 31 section headers, starting at offset 0x36d8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000000318 000318 00001c 00   A  0   0  1
  [ 2] .note.gnu.property NOTE            0000000000000338 000338 000020 00   A  0   0  8
  [ 3] .note.gnu.build-id NOTE            0000000000000358 000358 000024 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            000000000000037c 00037c 000020 00   A  0   0  4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0 0003a0 000024 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000000003c8 0003c8 0000a8 18   A  7   1  8
  [ 7] .dynstr           STRTAB          0000000000000470 000470 00008d 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          00000000000004fe 0004fe 00000e 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0000000000000510 000510 000030 00   A  7   1  8
  [10] .rela.dyn         RELA            0000000000000540 000540 0000f0 18   A  6   0  8
  [11] .rela.plt         RELA            0000000000000630 000630 000018 18  AI  6  24  8
  [12] .init             PROGBITS        0000000000001000 001000 000017 00  AX  0   0  4
  [13] .plt              PROGBITS        0000000000001020 001020 000020 10  AX  0   0 16
  [14] .plt.got          PROGBITS        0000000000001040 001040 000008 08  AX  0   0  8
  [15] .text             PROGBITS        0000000000001050 001050 00012f 00  AX  0   0 16
  [16] .fini             PROGBITS        0000000000001180 001180 000009 00  AX  0   0  4
  [17] .rodata           PROGBITS        0000000000002000 002000 000054 00   A  0   0  4
  [18] .eh_frame_hdr     PROGBITS        0000000000002054 002054 00003c 00   A  0   0  4
  [19] .eh_frame         PROGBITS        0000000000002090 002090 0000ec 00   A  0   0  8
  [20] .init_array       INIT_ARRAY      0000000000003dc0 002dc0 000010 08  WA  0   0  8
  [21] .fini_array       FINI_ARRAY      0000000000003dd0 002dd0 000010 08  WA  0   0  8
  [22] .dynamic          DYNAMIC         0000000000003de0 002de0 0001e0 10  WA  7   0  8
  [23] .got              PROGBITS        0000000000003fc0 002fc0 000028 08  WA  0   0  8
  [24] .got.plt          PROGBITS        0000000000003fe8 002fe8 000020 08  WA  0   0  8
  [25] .data             PROGBITS        0000000000004008 003008 000010 00  WA  0   0  8
  [26] .bss              NOBITS          0000000000004018 003018 000008 00  WA  0   0  1
  [27] .comment          PROGBITS        0000000000000000 003018 00001e 01  MS  0   0  1
  [28] .symtab           SYMTAB          0000000000000000 003038 000390 18     29  18  8
  [29] .strtab           STRTAB          0000000000000000 0033c8 0001f3 00      0   0  1
  [30] .shstrtab         STRTAB          0000000000000000 0035bb 00011a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

➢ Address of the ‘**.init**’ and ‘**.fini**’ sections are 1000 and 1180 by order. Now, we will be using objdump to disassemble the ‘**fini_init**’ program. We will be looking at the ‘.init’ and ‘.fini’ sections in order to compare the address values with the one we saw in the output of **readelf**.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ objdump --section .init -d fini_init

fini_init:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       48 83 ec 08             sub    $0x8,%rsp
    1004:       48 8b 05 c5 2f 00 00    mov    0x2fc5(%rip),%rax        # 3fd0 <__gmon_start__@Base>
    100b:       48 85 c0                test   %rax,%rax
    100e:       74 02                   je     1012 <_init+0x12>
    1010:       ff d0                   call   *%rax
    1012:       48 83 c4 08             add    $0x8,%rsp
    1016:       c3                      ret
```

➢ As we can see, '.init' section starts with 1000. It's the same address that we saw on the output of readelf command.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ objdump --section .fini -d fini_init

fini_init:     file format elf64-x86-64


Disassembly of section .fini:

0000000000001180 <_fini>:
    1180:       48 83 ec 08             sub    $0x8,%rsp
    1184:       48 83 c4 08             add    $0x8,%rsp
    1188:       c3                      ret
```

➢ '.fini' section starts with 1180. It's also the same address that we saw on the output of readelf command.

.text Section in ELF File

The **.text** section in an ELF File (compiled file) represents a special section that contains the executable instructions, also known as machine code. This is a core part of the program that the CPU can directly execute.

When we compile our source code (can be written in any programming language), our script turns into instructions, and it's placed under the '**.text**' section.

Let's create a very simple example.

```c
#include <stdio.h>

int main(){
    printf("Hello World!");
    return 0;
}
```

This is our source code. Let's compile it by using the syntax below.

gcc -o compiled_file_name source_code_name.c

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf --wide -S hello
There are 31 section headers, starting at offset 0x3690:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000000318 000318 00001c 00   A  0   0  1
  [ 2] .note.gnu.property NOTE            0000000000000338 000338 000020 00   A  0   0  8
  [ 3] .note.gnu.build-id NOTE            0000000000000358 000358 000024 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            000000000000037c 00037c 000020 00   A  0   0  4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0 0003a0 000024 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000000003c8 0003c8 0000a8 18   A  7   1  8
  [ 7] .dynstr           STRTAB          0000000000000470 000470 00008f 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          0000000000000500 000500 00000e 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0000000000000510 000510 000030 00   A  7   1  8
  [10] .rela.dyn         RELA            0000000000000540 000540 0000c0 18   A  6   0  8
  [11] .rela.plt         RELA            0000000000000600 000600 000018 18  AI  6  24  8
  [12] .init             PROGBITS        0000000000001000 001000 000017 00  AX  0   0  4
  [13] .plt              PROGBITS        0000000000001020 001020 000020 10  AX  0   0 16
  [14] .plt.got          PROGBITS        0000000000001040 001040 000008 08  AX  0   0  8
  [15] .text             PROGBITS        0000000000001050 001050 000108 00  AX  0   0 16
  [16] .fini             PROGBITS        0000000000001158 001158 000009 00  AX  0   0  4
  [17] .rodata           PROGBITS        0000000000002000 002000 000011 00   A  0   0  4
  [18] .eh_frame_hdr     PROGBITS        0000000000002014 002014 00002c 00   A  0   0  4
  [19] .eh_frame         PROGBITS        0000000000002040 002040 0000ac 00   A  0   0  8
  [20] .init_array       INIT_ARRAY      0000000000003dd0 002dd0 000008 08  WA  0   0  8
  [21] .fini_array       FINI_ARRAY      0000000000003dd8 002dd8 000008 08  WA  0   0  8
  [22] .dynamic          DYNAMIC         0000000000003de0 002de0 0001e0 10  WA  7   0  8
  [23] .got              PROGBITS        0000000000003fc0 002fc0 000028 08  WA  0   0  8
  [24] .got.plt          PROGBITS        0000000000003fe8 002fe8 000020 08  WA  0   0  8
  [25] .data             PROGBITS        0000000000004008 003008 000010 00  WA  0   0  8
  [26] .bss              NOBITS          0000000000004018 003018 000008 00  WA  0   0  1
  [27] .comment          PROGBITS        0000000000000000 003018 00001e 01  MS  0   0  1
  [28] .symtab           SYMTAB          0000000000000000 003038 000360 18     29  18  8
  [29] .strtab           STRTAB          0000000000000000 003398 0001dd 00      0   0  1
  [30] .shstrtab         STRTAB          0000000000000000 003575 00011a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

➢ As we can see, if we use readelf to see the sections in our ELF file, '**.text**' section starts with 1050.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ objdump --section .text -d hello

hello:     file format elf64-x86-64


Disassembly of section .text:

0000000000001050 <_start>:
    1050:       31 ed                   xor    %ebp,%ebp
    1052:       49 89 d1                mov    %rdx,%r9
    1055:       5e                      pop    %rsi
    1056:       48 89 e2                mov    %rsp,%rdx
    1059:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    105d:       50                      push   %rax
    105e:       54                      push   %rsp
    105f:       45 31 c0                xor    %r8d,%r8d
    1062:       31 c9                   xor    %ecx,%ecx
    1064:       48 8d 3d ce 00 00 00    lea    0xce(%rip),%rdi        # 1139 <main>
    106b:       ff 15 4f 2f 00 00       call   *0x2f4f(%rip)        # 3fc0 <__libc_start_main@GLIBC_2.34>
    1071:       f4                      hlt
    1072:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    1079:       00 00 00
    107c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000001080 <deregister_tm_clones>:
    1080:       48 8d 3d 91 2f 00 00    lea    0x2f91(%rip),%rdi        # 4018 <__TMC_END__>
    1087:       48 8d 05 8a 2f 00 00    lea    0x2f8a(%rip),%rax        # 4018 <__TMC_END__>
    108e:       48 39 f8                cmp    %rdi,%rax
    1091:       74 15                   je     10a8 <deregister_tm_clones+0x28>
    1093:       48 8b 05 2e 2f 00 00    mov    0x2f2e(%rip),%rax        # 3fc8 <_ITM_deregisterTMCloneTable@Base>
    109a:       48 85 c0                test   %rax,%rax
    109d:       74 09                   je     10a8 <deregister_tm_clones+0x28>
    109f:       ff e0                   jmp    *%rax
    10a1:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
    10a8:       c3                      ret
    10a9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000010b0 <register_tm_clones>:
    10b0:       48 8d 3d 61 2f 00 00    lea    0x2f61(%rip),%rdi        # 4018 <__TMC_END__>
    10b7:       48 8d 35 5a 2f 00 00    lea    0x2f5a(%rip),%rsi        # 4018 <__TMC_END__>
    10be:       48 29 fe                sub    %rdi,%rsi
    10c1:       48 89 f0                mov    %rsi,%rax
    10c4:       48 c1 ee 3f             shr    $0x3f,%rsi
    10c8:       48 c1 f8 03             sar    $0x3,%rax
    10cc:       48 01 c6                add    %rax,%rsi
    10cf:       48 d1 fe                sar    $1,%rsi
    10d2:       74 14                   je     10e8 <register_tm_clones+0x38>
    10d4:       48 8b 05 fd 2e 00 00    mov    0x2efd(%rip),%rax        # 3fd8 <_ITM_registerTMCloneTable@Base>
    10db:       48 85 c0                test   %rax,%rax
    10de:       74 08                   je     10e8 <register_tm_clones+0x38>
    10e0:       ff e0                   jmp    *%rax
    10e2:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    10e8:       c3                      ret
    10e9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000010f0 <__do_global_dtors_aux>:
    10f0:       f3 0f 1e fa             endbr64
    10f4:       80 3d 1d 2f 00 00 00    cmpb   $0x0,0x2f1d(%rip)        # 4018 <__TMC_END__>
    10fb:       75 2b                   jne    1128 <__do_global_dtors_aux+0x38>
    10fd:       55                      push   %rbp
    10fe:       48 83 3d da 2e 00 00    cmpq   $0x0,0x2eda(%rip)        # 3fe0 <__cxa_finalize@GLIBC_2.2.5>
    1105:       00
    1106:       48 89 e5                mov    %rsp,%rbp
    1109:       74 0c                   je     1117 <__do_global_dtors_aux+0x27>
    110b:       48 8b 3d fe 2e 00 00    mov    0x2efe(%rip),%rdi        # 4010 <__dso_handle>
    1112:       e8 29 ff ff ff          call   1040 <__cxa_finalize@plt>
    1117:       e8 64 ff ff ff          call   1080 <deregister_tm_clones>
    111c:       c6 05 f5 2e 00 00 01    movb   $0x1,0x2ef5(%rip)        # 4018 <__TMC_END__>
    1123:       5d                      pop    %rbp
    1124:       c3                      ret
    1125:       0f 1f 00                nopl   (%rax)
    1128:       c3                      ret
    1129:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001130 <frame_dummy>:
    1130:       f3 0f 1e fa             endbr64
    1134:       e9 77 ff ff ff          jmp    10b0 <register_tm_clones>

0000000000001139 <main>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       48 8d 05 c0 0e 00 00    lea    0xec0(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    1144:       48 89 c7                mov    %rax,%rdi
    1147:       b8 00 00 00 00          mov    $0x0,%eax
    114c:       e8 df fe ff ff          call   1030 <printf@plt>
    1151:       b8 00 00 00 00          mov    $0x0,%eax
    1156:       5d                      pop    %rbp
    1157:       c3                      ret
```

➢ There are two red marks on the image left-hand side, first one is **<_start>** and the second one is **<main>**. This situation may bring up a question: Why does the .text section start with **<_start>** instead of **<main>**?

Let's start talking about the main. Main is generally used in high-level languages such as C, C++. It represents the main operations of the program.

However, the <_start> tag is generally used in low-level languages such as Assembly. So, this <_start> tag specifies the starting point of compiled program.

- Let's analyze the '**.text**' section. Under the <_start> tag of the '**.text**' section, the <_start> tag calls the <main> function as shown in the image below. So, once the <_start> tag is triggered, it calls the <main> function as we can see on the '**lea**' *(load effective address)* line. As we can see, the '**rdi**', which is the destination index, specifies <main> function.

```
Disassembly of section .text:

0000000000001050 <_start>:
    1050:       31 ed                   xor    %ebp,%ebp
    1052:       49 89 d1                mov    %rdx,%r9
    1055:       5e                      pop    %rsi
    1056:       48 89 e2                mov    %rsp,%rdx
    1059:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    105d:       50                      push   %rax
    105e:       54                      push   %rsp
    105f:       45 31 c0                xor    %r8d,%r8d
    1062:       31 c9                   xor    %ecx,%ecx
    1064:       48 8d 3d ce 00 00 00    lea    0xce(%rip),%rdi        # 1139 <main>
    106b:       ff 15 4f 2f 00 00       call   *0x2f4f(%rip)        # 3fc0 <__libc_start_main@GLIBC_2.34>
    1071:       f4                      hlt
    1072:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    1079:       00 00 00
    107c:       0f 1f 40 00             nopl   0x0(%rax)
```

```
0000000000001139 <main>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       48 8d 05 c0 0e 00 00    lea    0xec0(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    1144:       48 89 c7                mov    %rax,%rdi
    1147:       b8 00 00 00 00          mov    $0x0,%eax
    114c:       e8 df fe ff ff          call   1030 <printf@plt>
    1151:       b8 00 00 00 00          mov    $0x0,%eax
    1156:       5d                      pop    %rbp
    1157:       c3                      ret
```

.plt (Procedure LinkageTable) and .got (Global Offset Table) Sections in ELF File

The '**.plt**' and '**.got**' sections deal with a large portion of the **dynamic linking**. The purpose of dynamic linking is that binaries do not have to carry all the code necessary to run within them. The '**.plt**' and '**.got**' sections work together to perform the linking.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf --wide -S hello
There are 31 section headers, starting at offset 0x3690:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000000318 000318 00001c 00   A  0   0  1
  [ 2] .note.gnu.property NOTE            0000000000000338 000338 000020 00   A  0   0  8
  [ 3] .note.gnu.build-id NOTE            0000000000000358 000358 000024 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            000000000000037c 00037c 000020 00   A  0   0  4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0 0003a0 000024 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000000003c8 0003c8 0000a8 18   A  7   1  8
  [ 7] .dynstr           STRTAB          0000000000000470 000470 00008f 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          0000000000000500 000500 00000e 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0000000000000510 000510 000030 00   A  7   1  8
  [10] .rela.dyn         RELA            0000000000000540 000540 0000c0 18   A  6   0  8
  [11] .rela.plt         RELA            0000000000000600 000600 000018 18  AI  6  24  8
  [12] .init             PROGBITS        0000000000001000 001000 000017 00  AX  0   0  4
  [13] .plt              PROGBITS        0000000000001020 001020 000020 10  AX  0   0 16
  [14] .plt.got          PROGBITS        0000000000001040 001040 000008 08  AX  0   0  8
  [15] .text             PROGBITS        0000000000001050 001050 000108 00  AX  0   0 16
  [16] .fini             PROGBITS        0000000000001158 001158 000009 00  AX  0   0  4
  [17] .rodata           PROGBITS        0000000000002000 002000 000011 00   A  0   0  4
  [18] .eh_frame_hdr     PROGBITS        0000000000002014 002014 00002c 00   A  0   0  4
  [19] .eh_frame         PROGBITS        0000000000002040 002040 0000ac 00   A  0   0  8
  [20] .init_array       INIT_ARRAY      0000000000003dd0 002dd0 000008 08  WA  0   0  8
  [21] .fini_array       FINI_ARRAY      0000000000003dd8 002dd8 000008 08  WA  0   0  8
  [22] .dynamic          DYNAMIC         0000000000003de0 002de0 0001e0 10  WA  7   0  8
  [23] .got              PROGBITS        0000000000003fc0 002fc0 000028 08  WA  0   0  8
  [24] .got.plt          PROGBITS        0000000000003fe8 002fe8 000020 08  WA  0   0  8
  [25] .data             PROGBITS        0000000000004008 003008 000010 00  WA  0   0  8
  [26] .bss              NOBITS          0000000000004018 003018 000008 00  WA  0   0  1
  [27] .comment          PROGBITS        0000000000000000 003018 00001e 01  MS  0   0  1
  [28] .symtab           SYMTAB          0000000000000000 003038 000360 18     29  18  8
  [29] .strtab           STRTAB          0000000000000000 003398 0001dd 00      0   0  1
  [30] .shstrtab         STRTAB          0000000000000000 003575 00011a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

Let's dump the '**.plt**' section and examine it.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ objdump -d -j .plt hello

hello:     file format elf64-x86-64


Disassembly of section .plt:

0000000000001020 <printf@plt-0x10>:
    1020:       ff 35 ca 2f 00 00       push   0x2fca(%rip)        # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 cc 2f 00 00       jmp    *0x2fcc(%rip)        # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000001030 <printf@plt>:
    1030:       ff 25 ca 2f 00 00       jmp    *0x2fca(%rip)        # 4000 <printf@GLIBC_2.2.5>
    1036:       68 00 00 00 00          push   $0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <_init+0x20>
```

- While examining the object dump of the '**.plt**' section, we can realize that **push** instruction pushes the value located at a memory address relative to the current instruction pointer (rip). In this case, the offset is **0x2fca**. This memory address points to a Global Offset Table entry that holds the address of the target function. Then, the **jmp** instruction tells the program to jump to another location within the same '**.plt**' section, specifically the address **#3ff8** which is also relative to the beginning of the section.

Now, it's time to dump the '**.got**' section and examine it.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ objdump -d -j .got hello

hello:     file format elf64-x86-64


Disassembly of section .got:

0000000000003fc0 <.got>:
        ...
```

- The '**.got**' section acts as a bridge between the program and the shared libraries. It's a table maintained within the program's memory space at runtime. The '.got' section contains the entry of **puts address** which is a standard C library used to print a string to the console.

If we consider our simple previous script, '**.text**' section cannot call the library directly in order to run the instructions because the compiler cannot know the address of the 'printf' function inside its library. For that reason, '**.text**' calls the '**.plt**' section to get routed to the address of the 'printf' function. So, it jumps from '**.plt**' to '**.got**' section to get the address entry of the 'printf' function.

Understanding these two sections will help us to perform some kind of attack such as **ret2plt**, **GOT overwrite** etc.

.rela.dyn And .rela.plt Sections in ELF File

The '**.rela.dyn**' and '**.rela.plt**' sections hold relocation entries that are essential for resolving symbol addresses at runtime.

In ELF Files, especially shared libraries (SOs), symbols (functions and variables) can reside in other libraries or the main executable. Relocation is the process of adjusting these symbol addresses during runtime to point to their actual memory locations. Relocation entries in sections like '**.rela.dyn**' and '**.rela.plt**' provide instructions for the dynamic linker to perform these adjustments.

.rela.dyn : This section contains relocation entries for *global dynamic symbols*. Global dynamic symbols are variables defined in a shared library and accessed by the main executable or other linked libraries at runtime. Entries in this section typically target the ***Global Offset Table (GOT)***. The dynamic linker uses the relocation information to modify the GOT (Global Offset Table) entries, ensuring they point to the correct memory address of the global dynamic symbols.

.rela.plt : This section contains relocation entries for functions accessed through the ***Procedure Linkage Table (PLT)***. Entries in '**.rela.plt**' typically target the *PLT stub*, a small piece of code within the PLT that performs the function address resolution. Dynamic Linker uses the relocation information to modify the PLT stub, ensuring it points to the correct function address. The PLT (Procedure Linkage Table) is a mechanism used in Position Independent Code (PIC) to resolve function addresses at runtime.

Position Independent Code (PIC): It allows code to be loaded at different memory addresses during execution.

Program Header in ELF File

The Program Header (Segment) is used by the Operating System and Dynamic Linker to locate the code and data in virtual memory. It provides instructions for the Operating System's loader on how to prepare the program execution in memory.

Keep in mind that Program Headers are essential for **executable** and **shared library files**, not *object files*.

The Program Header Table is an array of structures, each describing a single segment or other information. The size and number of entries are specified in the ELF Header itself. Program Header entries typically include **Segment Type**, **File Offset**, **Virtual Address**, **Segment Size in Memory**, **Segment Size in File**, **Permissions** and **Alignment**. Let's break those entries down one by one.

- **Segment Type:** It identifies the purpose of the segment (read-only data, loadable code, writable data).
- **File Offset:** It's the place where the segment's data resides within the ELF File.
- **Virtual Address:** The memory address where the segments should be loaded during the execution.
- **Segment Size in Memory:** The size of the segment in memory.

- **Segment Size in File:** The size of the segment's data within the ELF File.
- **Permissions:** It represents Read, Write and Executable permissions for the segment.
- **Alignment:** Memory alignment restrictions for the segment.

Let's take a look at those entries of the Program Header of our file and break it down once more.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis]
└─$ readelf -l hello

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1050
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000618 0x0000000000000618  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000161 0x0000000000000161  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000ec 0x00000000000000ec  R      0x1000
  LOAD           0x0000000000002dd0 0x0000000000003dd0 0x0000000000003dd0
                 0x0000000000000248 0x0000000000000250  RW     0x1000
  DYNAMIC        0x0000000000002de0 0x0000000000003de0 0x0000000000003de0
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000000358 0x0000000000000358
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x0000000000002014 0x0000000000002014 0x0000000000002014
                 0x000000000000002c 0x000000000000002c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002dd0 0x0000000000003dd0 0x0000000000003dd0
                 0x0000000000000230 0x0000000000000230  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00     
   01     .interp 
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt 
   03     .init .plt .plt.got .text .fini 
   04     .rodata .eh_frame_hdr .eh_frame 
   05     .init_array .fini_array .dynamic .got .got.plt .data .bss 
   06     .dynamic 
   07     .note.gnu.property 
   08     .note.gnu.build-id .note.ABI-tag 
   09     .note.gnu.property 
   10     .eh_frame_hdr 
   11     
   12     .init_array .fini_array .dynamic .got 
```

- ➢ **Type:** This type of program header specifies the type of segment the entry describes. There are various types like PHDR (Program Header Table itself), LOAD (Loadable Segment), DYNAMIC (Dynamic Linking Information) etc.
- ➢ **Offset:** This indicates the offset within the ELF file where the segment's data resides.

- **VirtAddr:** This shows the virtual address in memory where the segment will be loaded when the program executes.
- **MemSiz:** This represents the size in memory that the segment will occupy.
- **PhysAddr:** This is the Physical Address where the segment might be loaded in memory. However, it often has a special value like **0x0** for non-physical segments.
- **Flags:** These flags specify access permissions for the segment. R for Read; W for Write and X for Execute.
- **Align:** This indicates the alignment of requirement for the segment's virtual address.

What is Binary Analysis?

Binary Analysis is a process of examining the properties of binary files, including their containing instructions and data encoded in binary, in order to find out more about the content of file or purpose of the program. Binary analysis can play crucial roles in discovering bugs or security vulnerabilities.

- We can use different tools to perform Binary Analysis, such as Ghidra, Cutter, IDA, radare2, x64dbg (for Windows) etc.

Binary analysis can be categorized into two fundamental groups according to when it's been performed and also the level of complexity of the methods while performing the analysis.

Let's dive into the analysis types.

Static Analysis

Static Analysis involves examining the executable binary as it should be, but without running the executable itself. Static analysis tools may identify bugs or security vulnerabilities by only reading the program.

- Static Analysis consists of examining the executable file without viewing the actual instructions, hence it is straightforward and can be quick, but ineffective against large binaries and can miss important behaviors.
- Advanced static analysis consists of reverse engineering the binary's internals by loading the executable into a disassembler that transforms the binary code into human-readable text for looking at the program instructions or a decompiler that converts assembly-level idioms into high-level abstractions for examining the much more concise pseudocode, that typically omits some details to make the code easier for people to understand.

Dynamic Analysis

Dynamic Analysis involves observing the program as it executes in a real, or almost identical environment, if necessary, using full-system or user-mode emulation with QEMU. Dynamic analysis tools instrument the program with analysis code that stores information regarding the execution of the program as metadata.

- Basic dynamic analysis techniques involve running the binary and observing its behavior on the system, including the executed system and library calls and their arguments.

- ➢ Advanced dynamic analysis involves using a debugger to examine the internal state of a running executable, including the values of variables and the outcomes of conditional branches.

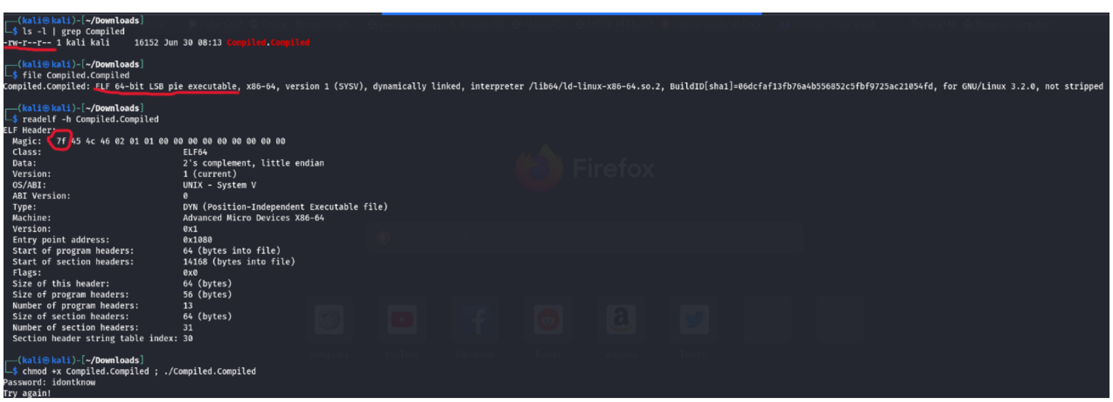
Analyzing ELF File

Compiled - TryHackMe

After covering the essentials about ELF File and Binary Analysis, let's try to analyze an ELF File. The file we will be analyzing can be found on TryHackMe website. This will be a simple example. You can access it from the **References** section. Let's dive right into it.

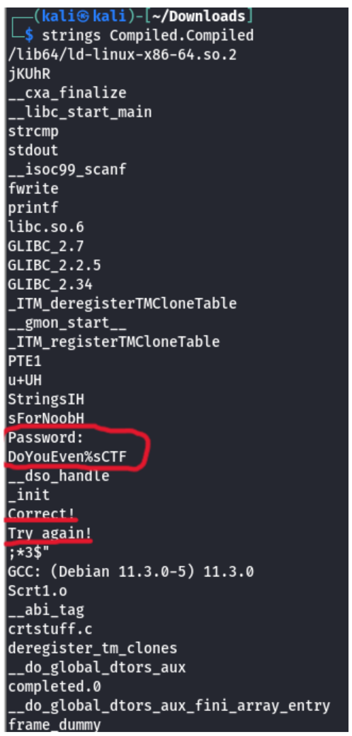
- ➢ First things first, let's download the ELF File from 'tryhackme.com'.

- ➢ As you can see, it asks us to find the password. So, let's begin.



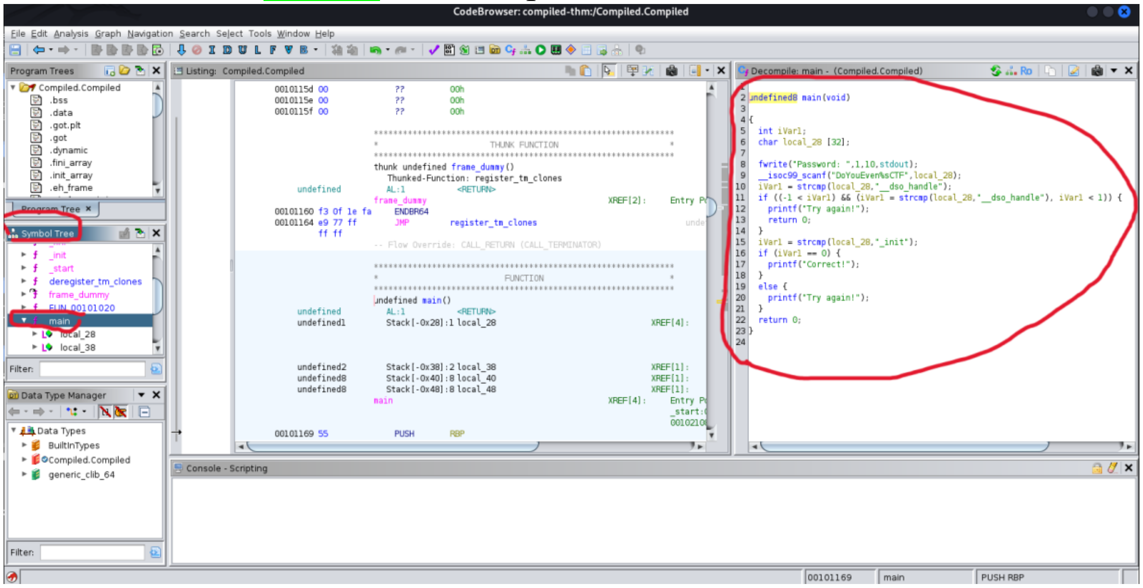
- ➢ In the first command, we are checking the permissions of our ELF File. As we can see, it doesn't have executable permission. Let us check if it's an

ELF file or not (This could be a text file). The **file** command helps us to identify if the file is ELF or not. Additionally, we can check it by using **readelf** command. As we can see on the image above, it's an ELF file. So, we should give the executable permission manually since it's an ELF file. We can give executable permission by entering the '***chmod +x Compiled.Compiled***' command. When we run the file, it asks for a password.



- **strings** command is used to extract the strings from our ELF File. While examining the output of '***string Compiled.Compiled***' command, we can see that there are some keywords (Underlined with red in the image). We can see that one of them is a success message, the other one is a failing message. On the other hand, we can also see the **Password** section and maybe the corresponding password, which is **DoYouEven%sCTF**.

- Now, let's use **Ghidra** to decompile our ELF file.

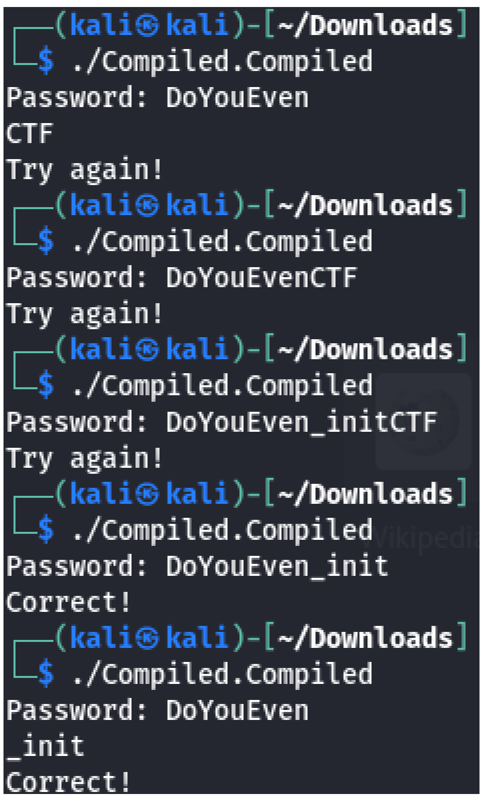


- After selecting the ELF file, and let Ghidra analyze it, we can access functions under **Symbol Tree** section on the left-hand side. When we click the **main** function, we can view the main function, as you can see on the right-hand side. Let's examine the main function.

```
undefined8 main(void)

{
  int iVar1;
  char local_28 [32];

  fwrite("Password: ",1,10,stdout);
  __isoc99_scanf("DoYouEven%sCTF",local_28);
  iVar1 = strcmp(local_28,"__dso_handle");
  if ((-1 < iVar1) && (iVar1 = strcmp(local_28,"__dso_handle"), iVar1 < 1)) {
    printf("Try again!");
    return 0;
  }
  iVar1 = strcmp(local_28,"_init");
  if (iVar1 == 0) {
    printf("Correct!");
  }
  else {
    printf("Try again!");
  }
  return 0;
}
```

- This script asks the user to enter the password, and then stores the input in variable called **local_28**. Then it compares the input string with **__dso_handle** and stores the result of comparison inside of a variable called **iVar1**. If **local_28** variable equals to **__dsohandle**, script will print 'Try Again!' message. After that, it compares the **local28** variable with **_init** and the result is assigned to a variable called **iVar1**. Finally, if **local_28** variable which is user input equals to '**_init**', script will print 'Correct!' message.

- Let's try to find the password. Considering our source code analysis, the correct password should contain '**_init**'. If we carefully look at the '**__isoc99_scanf("DoYouEven%sCTF",local_28);**' line, there should be a string in between DoYouEven and CTF. According to our findings, let's try to guess the password.

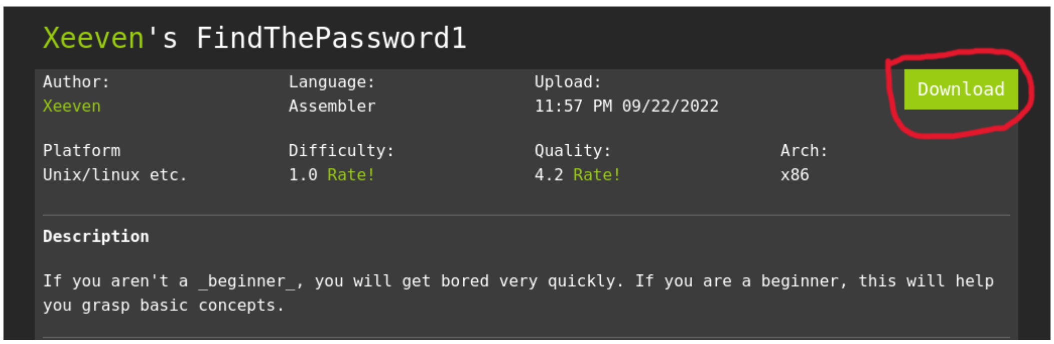


- In order to provide the '**local_28**' variable to equal to **_init**, there are two valid solutions, as you can see in the image. The first of them is **DouYouEven_init**, the other one is after filling with **DoYouEven** and hitting enter, it lets us enter another string which will be **_init**.

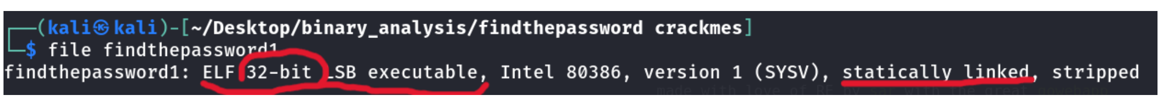
FindThePassword1 - Crackmes

The file we will be analyzing under this section can be found on 'crackmes.one' website. The name of the challenge is '**FindThePassword1**'. You can also access it from the **References** section. This will also be an easy challenge. Let's delve into the solution.

- Let's download the file from 'crackmes.one'.



- After downloading the file, it will require a password to extract the downloaded file. The password is '**crackmes.one**'. Let's check the type of our file in order to start analyzing it.



- As we can see from the output of '**file**' command, structure of this file is a little bit different than the previous one. It's **32-bit** executable (x86) file and it's **statically linked** while the previous one was **64-bit** and **dynamically linked**. Since it's an executable, if it does not have executable permission, we can give it by using '**chmod +x findthepassword1**' command.

Let's talk about the difference between static and dynamic linking.

- In **Statically Linked files**, all required libraries to run the file are embedded directly into the executable file and this feature allows file to run on any system with a compatible architecture but the size of file is larger.
- In **Dynamically Linked files**, only the references (symbols) to the required libraries are embedded in the executable file, which provides smaller size. The executable file relies on the presence of the required libraries on the target system, missing libraries causes the program to fail.

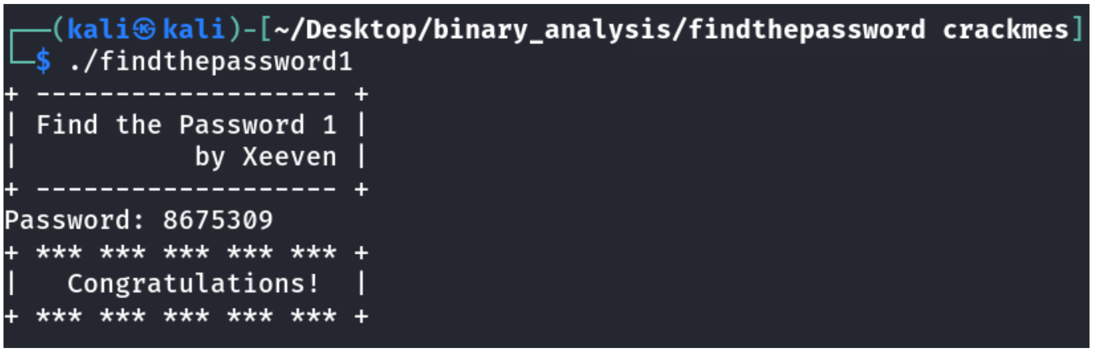
Now, let's see what happens when we run the program.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/findthepassword crackmes]
└─$ ./findthepassword1
+ ------------------- +
| Find the Password 1 |
|          by Xeeven |
+ ------------------- +
Password: topsecret
Wrong! Try again.
```

- The program asks for a password, as its name suggests. Let's dump the strings of our executable file by using the **strings** command.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/findthepassword crackmes]
└─$ strings findthepassword1
+ ------------------- +
| Find the Password 1 |
|          by Xeeven |
+ ------------------- +
Password: + *** *** *** *** *** +
|   Congratulations!  |
+ *** *** *** *** *** +
Wrong! Try again.
8675309
.shstrtab
.text
.data
.bss
```

- As we can see from the dump of the strings, there is a possible match of the password, which is **8675309**. Let's give it a try to find out.

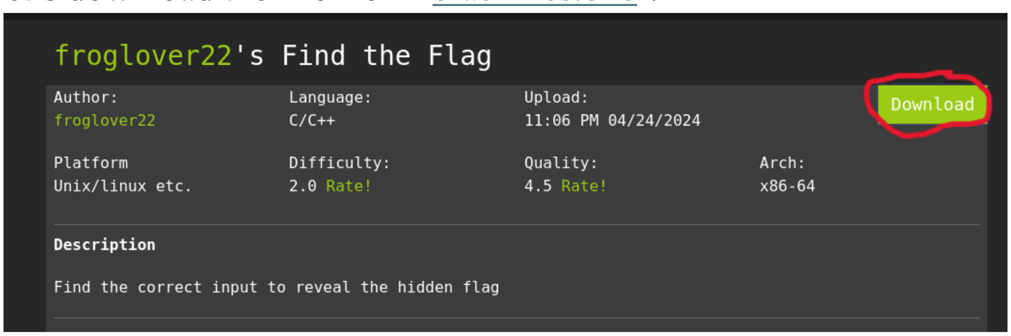


- As the image proves, it worked.

Find The Flag - Crackmes

The file we will be analyzing under this section can also be found on '[crackmes.one]' website. The name of the challenge is '**Find The Flag**'. You can also access it from the **References** section. This challenge is not easy as previous ones. We will try new approaches in this challenge. Let's delve into the solution.

- Let's download the file from '[crackmes.one]'.



- After downloading the file, it will require a password to extract the downloaded file as the previous one. The password is also **crackmes.one**. Let's check the type of our file in order to start analyzing it.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/find_the_flag crackmes]
└─$ file two
two: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e31572cd709d1bf78390c4c32dc7cb646bf5db87, for GNU/Linux 3.2.0, with debug_info, not stripped
```

- As seen in the image above, our file is **64-bit** executable and **dynamically linked**. Refer to previous section to learn more about difference between dynamic and static linking.
- As a reminder, there is one more way to check if the structure of the file is ELF or not. We will do it by dumping hex values of the file. So, **hexdump** tool will be used. Let's take a look how we can use it.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/find_the_flag crackmes]
└─$ hexdump -C two | head -n 3
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  90 10 00 00 00 00 00 00  |..>.............|
00000020  40 00 00 00 00 00 00 00  80 3d 00 00 00 00 00 00  |@........=......|
```

- In this code snippet, **-C** parameter dumps hex values of the file and also displays ASCII texts. Since our purpose is to see if the file is ELF or not, we need to see the file header. So, we use piping (|) and **head -n 3** command to see the first three lines of the dump.
- As previously mentioned, ELF Header has a sequence of four unique bytes that are **0x7F** followed by **0x45**, **0x4c**, and **0x46**. Our file starts with these series of bytes, which proves that our file is an ELF file.

Let's run the file and see what happens.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/find_the_flag crackmes]
└─$ ./two
sh: 1: %cjpka%qm: not found
Incorrect.
```

- In order to find the flag, we have to provide the correct input. It's time to start analyzing the file.

```
int main(int argc,char **argv)

{
  int iVar1;
  char local_a8 [8];
  char cmd [80];
  char shell [39];
  int local_24;
  int local_20;
  int j;
  int i;
  int bitmask;
  char **argv_local;
  int argc_local;

  j = 0x14;
  if (argc == 2) {
    j = atoi(argv[1]);
  }
  for (local_20 = 0; local_20 < 5; local_20 = local_20 + 1) {
    if ((j & 1 << ((byte)local_20 & 0x1f)) != 0) {
      for (local_24 = 0; local_24 < 8; local_24 = local_24 + 1) {
        chunks[local_20][local_24] = chunks[local_20][local_24] ^ 5;
      }
    }
  }
  memset(cmd + 0x48,0,0x27);
  snprintf(cmd + 0x48,0x27,"%s%s%s%s%s",chunk_one,chunk_two,chunk_three,chunk_four,chunk_five);
  memset(local_a8,0,0x50);
  snprintf(local_a8,0x50,"sh -c \'%s\' 2>/dev/null",cmd + 0x48);
  iVar1 = system(local_a8);
  if (iVar1 != 0) {
    puts("Incorrect.");
  }
  return 0;
}
```

- This is the main function. Let's start examining the main function. First of all, the value of j variable (0x14) corresponds to 20 in decimal [(1*16^1) + (4*16^0) = 20]. Keep in mind that 0x notation represents hexadecimal values (base16). Let's continue with the if statement.

```
j = 0x14;
if (argc == 2) {
  j = atoi(argv[1]);
}
```

➢ This code snippet initializes j variable (previously defined above), and basically checks a condition. Before understanding what this condition does, let's understand the **argc** function. It's an integer variable that holds the number of arguments passed to the program when it's executed. In this case, the if statement checks if the program was called with two arguments. (including the program itself). For instance, if there's only the program name, argc will be 1. Therefore, the if statement will be **false**. Inside of the if statement, let's clarify the **atoi** function. It converts a string representation of an integer to its actual integer value. The **argv[1]** refers to the second argument passed to the program when it was executed (argv[0] would be the program itself). What we should conclude from this if statement, we need to pass an argument when executing the program. Now, let's focus on how we can find the accurate argument(s). The next thing to examine is the for loop.

```
for (local_20 = 0; local_20 < 5; local_20 = local_20 + 1) {
  if ((j & 1 << ((byte)local_20 & 0x1f)) != 0) {
    for (local_24 = 0; local_24 < 8; local_24 = local_24 + 1) {
      chunks[local_20][local_24] = chunks[local_20][local_24] ^ 5;
    }
  }
}
```

➢ This for loop iterates five times, with **local_20** variable taking values from 0 to 4. The if statement inside of the for loop responsible for determining whether the inner loop runs for a specific row (local_20). Let's try to explain it in detail.

- **1 <<((byte)local_20 & 0x1f):** This might look confusing. Let's break it down. **(byte)local_20** parameter casts local_20 variable to a byte (8 bits) to ensure it doesn't exceed the valid range for bitwise operations. **& 0x1f** parameter performs a bitwise AND operation between **local_20** (as a byte) and **0x1f** (31 in decimal). This one essentially isolates the lower 5 bits of **local_20**.

- **1 <<:** This left-shifts a 1 by value obtained after the AND operation. So, for **local_20 = 0**, the shift is by 0 (resulting in 1). For **local_20 = 1**, the shift is by 1 (resulting in 2), and so on. This creates a mask where a single bit is set at a specific position based on the value of **local_20**.
- **j &:** This performs a bitwise AND operation between j (which is 20) and the left-shifted value. It checks if the corresponding bit in j and the mask is set to 1.
- **!=:** The whole condition is about to check if the result of the AND operation is not zero. If a specific bit in j (based on local_20) is set, the condition is true, and the inner loop executes for that row.

➢ The default value of **j** variable is **0x14** which is 20 in decimal. However, if there is an argument, the value of j variable will be changed according to the given argument.

Now, let's examine the rest of the script. It will be on the next page to see it clearly.

```
memset(cmd + 0x48,0,0x27);
snprintf(cmd + 0x48,0x27,"%s%s%s%s%s",chunk_one,chunk_two,chunk_three,chunk_four,chunk_five);
memset(local_a8,0,0x50);
snprintf(local_a8,0x50,"sh -c \'%s\' 2>/dev/null",cmd + 0x48);
iVar1 = system(local_a8);
if (iVar1 != 0) {
  puts("Incorrect.");
}
return 0;
}
```

- We will be examining the lines demonstrated above. We will be covering line by line.
  - ***memset(cmd + 0x48, 0, 0x27):*** Generally speaking, this line prepares a buffer for storing a command string. The **memset** function fills a block of memory with a specific value. Let's clarify the **cmd + 0x48** argument points to a memory location within the previously defined **cmd array**, offset by **0x48** (72 in decimal). This might be allocating space for the command string after some initial data in cmd. The **0** (zero) value right after 0x48 specifies the value to fill the memory with (in this case, null characters). Finally, **0x27** defines the number of bytes to fill (39 in decimal). It likely reserves space for the command string (including the null terminator).
  - ***snprintf(cmd + 0x48, 0x27, "%s%s%s%s%s", chunk_one, chunk_two, chunk_tree, chunk_four, chunk_five):*** In general, this line constructs the actual command string within the cmd buffer. The **snprinft** function formats and writes a string to a buffer with a limited size in order to prevent buffer overflow attacks. **cmd + 0x48, 0x27** is the same thing as previous line. **"%s%s%s%s%s"** is a format string that defines how the following arguments will be inserted into the resulting string. In this case, **chunk_one, chunk_two, chunk_three, chunk_four, chunk_five** variables are inserted. These are assumed to

be variables containing string values that will be inserted into the format string at the corresponding positions.

- ***memset(local_a8, 0, 0x50):*** This line prepares another buffer (local_a8) for storing a command string in general. We've covered what **memset** function does previously. The **local_a8** variable points to the memory location for the **local_a8** array. **0** (zero) value fills the memory with null characters. **0x50** reserves space for the final command string (80 in decimal).
- ***snprint(local_a8, 0x50, "sh -c \ '%s' 2>/dev/null", cmd + 0x48):*** In general, this line constructs the final command string to be executed within **local_a8**. Since we previously talked about the **snprintf** function, we won't be talking about it in this part. The **local_a8** variable points to the buffer where the formatted string will be written. **0x50** represents the maximum number of bytes to write. **"%s -c \ '%s\' 2>/dev/null"** is the format string which defines how the arguments will be inserted. Let's take a look at these arguments. **sh -c** refers Bash (Bourne Again Shell) which is a shell programming language and **-c** parameter tells Bash to take the following argument as a command to execute. **\ '%s\'** is a placeholder for another string enclosed within single quotes. **2>/dev/null** redirects standard error messages to /dev/null, which is like a black hole in Linux file system. Lastly, **cmd + 0x48** argument is inserted into the placeholder within single quotes. It references the location within cmd where the previously constructed command string is stored.
- ***iVar1 = system(local_a8):*** This line executes the command string constructed in **local_a8** using the system function. **system** function takes a string argument representing a command to be executed by the

system shell. **local_a8** variable responsible for holding the final command string that combines the previously constructed command from **chunk_one** to **chunk_five** with the shell invocation and error redirection. **iVar1** variable stores the return value of the **system** function.

- ***if (iVar1 != 0){ puts("Incorrect."); }:*** This if statement checks the return value of the system function stored in iVar1. **iVar1 != 0** condition checks if the return value is not zero. **puts("Incorrect")** prints a string to the standard output (usually the console).

As we've analyzed the main function, let's think about what we can do in order to find the flag. According to the main function, the default value of j is 0x14, which is 20 in decimal. However, there is a point that we need to pay attention to. If there is a provided input argument, value of j variable will get changed based on that argument. Each bit in j variable determines whether certain chunks will be modified with XOR operation of 5. If the result is not zero, it prints Incorrect.

Here are the chunk values.

```
LEA    RCX,[chunk_one]      = "`fmj%'Fj"
LEA    R8,[chunk_two]       = "kbwdqv$%"
LEA    R9,[chunk_three]     = "Youve fo"
LEA    R10,[chunk_four]     = "pka%qm`%"
LEA    RAX,[chunk_five]     = "flag.\""
```

chunk_one = "`fmj%'Fj"

chunk_two = "kbwdqv$%"

chunk_three = "Youve fo"

chunk_four = "pka%qm`%"

chunk_five = "flag.\""

Now, let's try to decrypt these chunks. We will be doing it by writing a script. It will be on the next page.

```python
chunks = ["`fmj%'Fj", "kbwdqv$%", "Youve fo", "pka%qm`%", "flag.\""]

def xor_decrypt(chunk_value, xor_key): # --> XOR decryption
    global empty_str
    empty_str = ''
    return empty_str.join(chr(ord(character) ^ xor_key) for character in chunk_value)
#ord() --> integer representation of the character's ASCII,
# ^ --> bitwise operation between character's code and key
#chr() --> convert integer to char

def generate_chunks(x): # --> x as input to modify tge original chunks
    modified_list = [] #empty list
    for i, chunk in enumerate(chunks): # iterating over chunks list with i index.
        if x & (1 << i): # checking if the x value has a bit set corresponding 1 << i
            modified_list.append(xor_decrypt(chunk, 5)) #appending chunks and chunk_key as 5
        else:
            modified_list.append(chunk) #original chunk appending to the modeified_chunk without modification
    return modified_list

for x in range(1, 50): #iterating values from 1 to 49
    modified_list = generate_chunks(x) #calling function with x value
    decrypted_chunks = empty_str.join(modified_list) #joining into a single string
    print(f'For x = {x} , FLAG: {decrypted_chunks}') #printing current value of x and decrypted_chunks
```

- This script is dedicated to decrypting the XOR chunk values we've found while analyzing the executable file. Comment lines demonstrate what the corresponding line is doing. Let's run this script to find the value corresponding to decrypted message.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/find_the_flag crackmes]
└─$ python decode_xor.py
For x = 1 , FLAG: echo "Cokbwdqv$%Youve fopka%qm`%flag."
For x = 2 , FLAG: `fmj%'Fjngrats! Youve fopka%qm`%flag."
For x = 3 , FLAG: echo "Congrats! Youve fopka%qm`%flag."
For x = 4 , FLAG: `fmj%'Fjkbwdqv$%\jps`%cjpka%qm`%flag."
For x = 5 , FLAG: echo "Cokbwdqv$%\jps`%cjpka%qm`%flag."
For x = 6 , FLAG: `fmj%'Fjngrats! \jps`%cjpka%qm`%flag."
For x = 7 , FLAG: echo "Congrats! \jps`%cjpka%qm`%flag."
For x = 8 , FLAG: `fmj%'Fjkbwdqv$%Youve found the flag."
For x = 9 , FLAG: echo "Cokbwdqv$%Youve found the flag."
For x = 10 , FLAG: `fmj%'Fjngrats! Youve found the flag."
For x = 11 , FLAG: echo "Congrats! Youve found the flag."
For x = 12 , FLAG: `fmj%'Fjkbwdqv$%\jps`%cjund the flag."
For x = 13 , FLAG: echo "Cokbwdqv$%\jps`%cjund the flag."
For x = 14 , FLAG: `fmj%'Fjngrats! \jps`%cjund the flag."
For x = 15 , FLAG: echo "Congrats! \jps`%cjund the flag."
For x = 16 , FLAG: `fmj%'Fjkbwdqv$%Youve fopka%qm`%cidb+'
For x = 17 , FLAG: echo "Cokbwdqv$%Youve fopka%qm`%cidb+'
For x = 18 , FLAG: `fmj%'Fjngrats! Youve fopka%qm`%cidb+'
For x = 19 , FLAG: echo "Congrats! Youve fopka%qm`%cidb+'
For x = 20 , FLAG: `fmj%'Fjkbwdqv$%\jps`%cjpka%qm`%cidb+'
For x = 21 , FLAG: echo "Cokbwdqv$%\jps`%cjpka%qm`%cidb+'
For x = 22 , FLAG: `fmj%'Fjngrats! \jps`%cjpka%qm`%cidb+'
For x = 23 , FLAG: echo "Congrats! \jps`%cjpka%qm`%cidb+'
For x = 24 , FLAG: `fmj%'Fjkbwdqv$%Youve found the cidb+'
For x = 25 , FLAG: echo "Cokbwdqv$%Youve found the cidb+'
For x = 26 , FLAG: `fmj%'Fjngrats! Youve found the cidb+'
For x = 27 , FLAG: echo "Congrats! Youve found the cidb+'
For x = 28 , FLAG: `fmj%'Fjkbwdqv$%\jps`%cjund the cidb+'
For x = 29 , FLAG: echo "Cokbwdqv$%\jps`%cjund the cidb+'
For x = 30 , FLAG: `fmj%'Fjngrats! \jps`%cjund the cidb+'
For x = 31 , FLAG: echo "Congrats! \jps`%cjund the cidb+'
For x = 32 , FLAG: `fmj%'Fjkbwdqv$%Youve fopka%qm`%flag."
For x = 33 , FLAG: echo "Cokbwdqv$%Youve fopka%qm`%flag."
For x = 34 , FLAG: `fmj%'Fjngrats! Youve fopka%qm`%flag."
For x = 35 , FLAG: echo "Congrats! Youve fopka%qm`%flag."
For x = 36 , FLAG: `fmj%'Fjkbwdqv$%\jps`%cjpka%qm`%flag."
For x = 37 , FLAG: echo "Cokbwdqv$%\jps`%cjpka%qm`%flag."
For x = 38 , FLAG: `fmj%'Fjngrats! \jps`%cjpka%qm`%flag."
For x = 39 , FLAG: echo "Congrats! \jps`%cjpka%qm`%flag."
For x = 40 , FLAG: `fmj%'Fjkbwdqv$%Youve found the flag."
For x = 41 , FLAG: echo "Cokbwdqv$%Youve found the flag."
For x = 42 , FLAG: `fmj%'Fjngrats! Youve found the flag."
For x = 43 , FLAG: echo "Congrats! Youve found the flag."
For x = 44 , FLAG: `fmj%'Fjkbwdqv$%\jps`%cjund the flag."
For x = 45 , FLAG: echo "Cokbwdqv$%\jps`%cjund the flag."
For x = 46 , FLAG: `fmj%'Fjngrats! \jps`%cjund the flag."
For x = 47 , FLAG: echo "Congrats! \jps`%cjund the flag."
For x = 48 , FLAG: `fmj%'Fjkbwdqv$%Youve fopka%qm`%cidb+'
For x = 49 , FLAG: echo "Cokbwdqv$%Youve fopka%qm`%cidb+'
```

- As the output shows us, the values we're looking for are **11** and **43**, if the range is between 1 to 49. Of course, we can extend this range to whatever we'd like to.

```
┌──(kali㉿kali)-[~/Desktop/binary_analysis/find_the_flag crackmes]
└─$ ./two 11
Congrats! Youve found the flag.

┌──(kali㉿kali)-[~/Desktop/binary_analysis/find_the_flag crackmes]
└─$ ./two 43
Congrats! Youve found the flag.
```

➢ After putting a little bit of effort, we're delighted to find the flag.

References:

https://www.udemy.com/course/linux-binary-analizi/

https://www.geeksforgeeks.org/cc-preprocessors/

https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html

https://stackoverflow.com/questions/7718299/whats-an-object-file-in-c

https://stackoverflow.com/questions/15284947/understanding-gcc-s-output

https://www.baeldung.com/linux/executable-and-linkable-format-file

https://medium.com/@boutnaru/linux-elf-executable-and-linkable-format-part-1-intro-47bd61af105e

https://docs.oracle.com/cd/E19455-01/806-3773/elf-2/index.html

https://ftp.math.utah.edu/u/ma/hohn/linux/misc/elf/node3.html

https://www.youtube.com/watch?v=B4-wVdQo040

https://ir0nstone.gitbook.io/notes/types/stack/aslr/plt_and_got

https://bugprove.com/knowledge-hub/binary-analysis-fundamentals/

https://www.chromethemer.com/backgrounds/phone/binary-code-phone-background-wallpaper.html

https://tryhackme.com/r/room/compiled

https://crackmes.one/crackme/632cf67b33c5d4425e2cd501

https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/