

Linux Forensics

Binary analysis – Part2

Executable and **L**inkable **F**ormat
Approach



www.vistacompany.ir

WHOAMI

Seyyed Hossein Kamali



- ❑ Team Lead, Cyber Threat Defense Centre@ Dotin
- ❑ Main focus on hunting badness

Executable and Linkable Format

Why Learning ELF Files?

Why should we know about ELF files?

- ELF file analysis used by:
- ❖ Blue Teamers
 - ❖ Incident Response
 - ❖ Digital Forensics
 - ❖ Malware Researchers
 - ❖ Red Teamers

Definition

ELF (**Executable and Linkable Format**) formerly named **Extensible Linking Format** standard file format for **executable files**, **object code**, **shared libraries**, and **core dumps**. First introduced with Unix system and is now standard executable file format on Linux, FreeBSD and any other device like micro controller and many other things. By design, the ELF format is flexible, extensible, and cross-platform. For instance it supports different endiannesses and address sizes so it does not exclude any particular central processing unit (CPU) or instruction set architecture.(7)

ELF Extension

ELF known file extensions

.axf | .bin | .elf | .o | .prx | .puff | .ko | .mod | .so

Anatomy of an Executable File

Each operating systems have two
fundamental abstractions



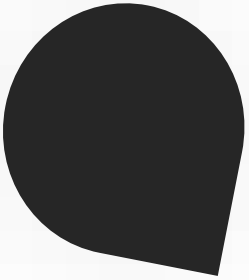
Processes

processes can be viewed as
a dynamic representation of
resources.



Files

Binary or executable files can be
viewed as static representation of
resources



Definition

The process of transforming the static object (*binary executable files*) in a dynamic object (*process*) is called **loading**.

ELF files are used by
two tools

Linker and Loader

Linker: *Linking* is the process of combining various pieces of code and data together to form a single executable that can be loaded in memory. Linking can be done at compile time, at load time (by loaders) and also at run time (by application programs).

Loader: The *loader* is a program called **execve**, which loads the code and data of the executable object file into memory and then runs the program by jumping to the first instruction.(3)

- in real environments, with dynamic linking, loading may require relocation. Why?
 - Because, if the file is dynamically linked it has to be linked again with all the shared libraries it depends on.

In the next part, I'll describe the full relocation and symbol resolution structure.

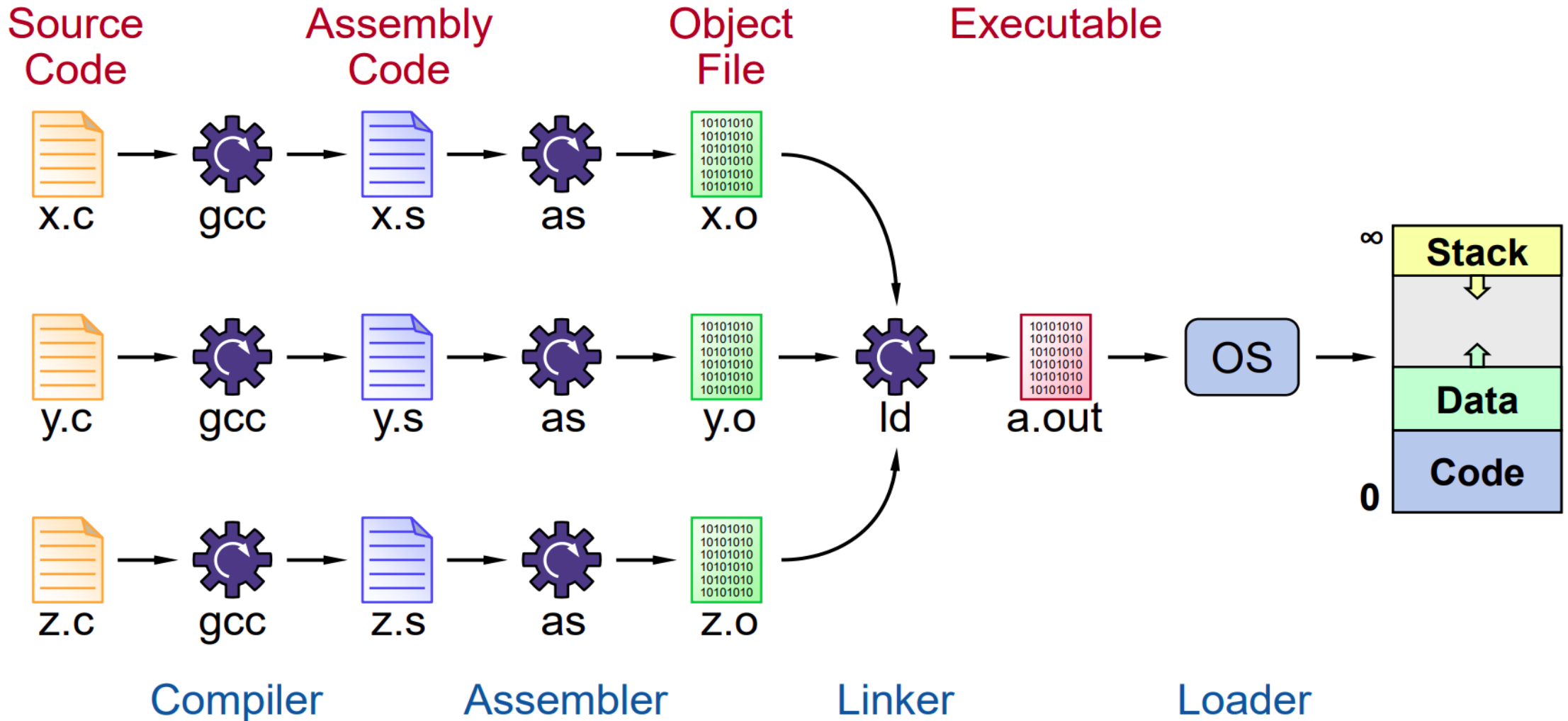
Linkers and loaders perform various related but conceptually different tasks:

Linker and Loader

- **Program Loading:** This refers to copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state. In some cases, program loading also might involve allocating storage space or mapping virtual addresses to disk pages.
- **Relocation:** Compilers and assemblers generate the object code for each input module with a starting address of zero. Relocation is the process of assigning load addresses to different parts of the program by merging all sections of the same type into one section. The code and data section also are adjusted so they point to the correct runtime addresses.
- **Symbol Resolution:** A program is made up of multiple subprograms; reference of one subprogram to another is made through symbols. A linker's job is to resolve the reference by noting the symbol's location and patching the caller's object code.(3)

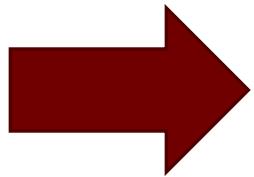
In the next part, I'll describe the full relocation and symbol resolution structure.

Creating a Process (borrow from index 5 of ref.)



An ELF file provides 2 views on the data, contains₍₄₎

Loading View

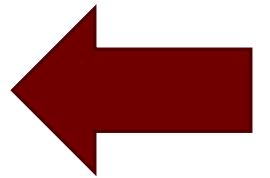


ELF Header
Program Header Table
Segment 1
Segment 2
...
Section Header Table <i>optional</i>

Only the ELF header has a fixed position in the file. The flexibility of the ELF format requires no specified order for header tables, sections or segments.

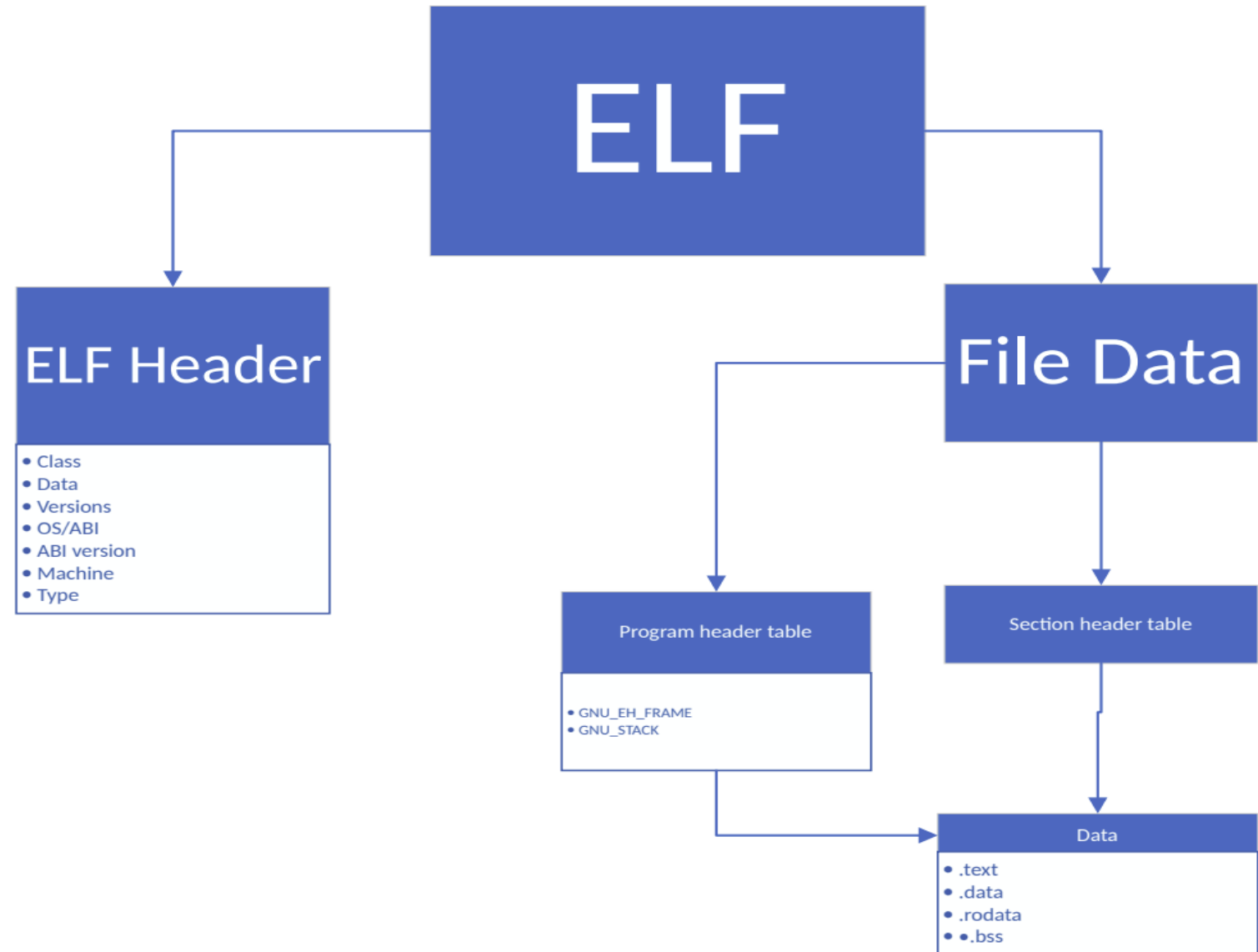
ELF Header
Program Header Table <i>optional</i>
Section 1
...
Section <i>n</i>
...
...
Section Header Table

Linking View



ELF Views

Simplified version of
the structure of an
ELF-file.(2)



Write Sample Program

I'll write a test program with c languages and make elf format. Then I decided to analyze it.

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

int main (int argc, char **argv){
    char buf[128];
    if(argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

ELF Header

-> gcc 484_test.c -o
484_test

The ELF file
header tells where
program header
table & section
header table are.

```
root@slingshot:/home# readelf -h 484b_test
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 (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                             0x1
  Entry point address:                 0x5f0
  Start of program headers:            64 (bytes into file)
  Start of section headers:           6544 (bytes into file)
  Flags:                               0x0
  Size of this header:                  64 (bytes)
  Size of program headers:              56 (bytes)
  Number of program headers:            9
  Size of section headers:              64 (bytes)
  Number of section headers:            29
  Section header string table index:    28
```

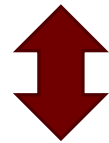

ELF Header Inspecting...

ELF Header:

Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00



```
root@slingshot:/home# hexdump -C 484b_test
00000000  7f 45 4c 46 |.ELF
```



```
root@slingshot:/home# od -t x1 -c 484b_test
```

```
00000000  7f  45  4c  46
          177   E   L   F
```



Padding Bytes.
These bytes are
unused and are
always set to 0

ELF Header Inspecting...

- Class:
 - ❖ x64 -> (=02)
 - ❖ x32 -> (=01)
- Data:
 - ❖ LSB -> (=01)
 - ❖ MSB -> (=02)
- Version:
 - ❖ Current Version -> (=01)
 - ❖ Invalid Version -> (=02)

ELF Header Inspecting...

- OS/ ABI : An Application Binary Interface (ABI) is the interface between two binary program modules that work together. An ABI is a contract between pieces of binary code defining the mechanisms by which functions are invoked and how parameters are passed between the caller and callee.
- ABI Version: Show which version of ABI used.

ELF Header Inspecting...

- Type:
 - ❖ Relocation File -> (=01)
 - ❖ Executable File -> (=02)
 - ❖ Shared Object File -> (=03)
 - ❖ Core File -> (=04)
- Machine: denotes the architecture that the binary is intended to run on.

ELF Header Inspecting...

- Entry Point Address: where does the program start?
- Start of Program Header: Identifies the start of the program headers with bytes into the ELF-file.
- Start of Section Header: Identifies the start of the section headers with bytes into the ELF-file.
- Size of Program Header: Identifies the size of the program headers that is in the ELF-file.

ELF Header Inspecting...

- Size of section headers: Identifies the size of the section headers that is in the ELF-file.
- Number of program headers: Identifies how many program headers there is in the ELF-file.
- Number of section headers: Identifies how many section headers there is in the ELF-file.(2)

Run The Program

Run the program and
auditd log:

```
root@slingshot:/home# ./484b_test Hello_World
Hello_World
```

Syscall - > execve

Syscall - > execve

```
type=SYSCALL msg=audit(1655365953.222:301): arch=c000003e syscall=59 success=yes exit=0 a0=564c5
5133010 a1=564c55135320 a2=564c550afa50 a3=8 items=2 ppid=23566 pid=23769 auid=1001 uid=0 gid=0
euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts1 ses=50 comm="484b_test" exe="/home/484b_tes
t" key=(null)
type=EXECVE msg=audit(1655365953.222:301): argc=2 a0="./484b_test" a1="Hello_World"
type=CWD msg=audit(1655365953.222:301): cwd="/home"
type=PATH msg=audit(1655365953.222:301): item=0 name="./484b_test" inode=3426972 dev=08:03 mode=
0100755 ouid=0 ogid=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000
cap_fe=0 cap_fver=0
type=PATH msg=audit(1655365953.222:301): item=1 name="/lib64/ld-linux-x86-64.so.2" inode=6946839
dev=08:03 mode=0100755 ouid=0 ogid=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=
0000000000000000 cap_fe=0 cap_fver=0
type=PROCTITLE msg=audit(1655365953.222:301): proctitle=2E2F343834625F746573740048656C6C6F5F576F
726C64
```

Path of Binary execution

Interpreter Call

Type of ELF Files

- Binary executable (ET_EXEC)
 - Runnable program, must have Segments
- Object files (or relocatable objects (.o), ET_REL)
 - Links with other object files, must have sections.
- Shared Library (.so, ET_DYN)
 - Links with other object files/executables.
 - Has both segments and sections.
- Core Dump(ET_CORE)
 - Generated when program receives SIGABRT.
 - Has no sections, has segments(PT_LOAD/ PT_Notes)

I'll discuss and analysis
all type of elf files in next
parts.

Define Segment and Section

Sections comprise all information needed for linking a target object file in order to build a working executable. In the other word, **Sections** represent the smallest indivisible units that can be processed within an ELF file. sections perspective of a linker.

Segments, which are commonly known as Program Headers, break down the structure of an ELF binary into suitable chunks to prepare the executable to be loaded into memory.(borrow from Intezer web site)

Segments are a collection of sections that represent the smallest individual units that can be mapped to a memory image by the runtime linker.

Sections hold the bulk of object file information for the linking view:
instructions, data, symbol table, relocation information, and so on.

Program Header Table

- A program header table is an array of program headers that defines the memory layout of a program at runtime.
- The program header shows the segments used at runtime, and tells the system how to create a process image. An ELF-file can consist of zero or more segments. The kernel can access the segments and map them into a virtual address space by using mmap system calls. (6)

Elf file type is DYN (Shared object file)
Entry point 0x5f0
There are 9 program headers, starting at offset 64

Program Headers:

Next Part...

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
1 PHDR	0x0000000000000040 0x00000000000001f8	0x0000000000000040 0x00000000000001f8	0x0000000000000040 R 0x8
2 INTERP	0x0000000000000238 0x000000000000001c	0x0000000000000238 0x000000000000001c	0x0000000000000238 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
3 LOAD	0x0000000000000000 0x0000000000000948	0x0000000000000000 0x0000000000000948	0x0000000000000000 R E 0x200000
LOAD	0x0000000000000da8 0x0000000000000268	0x0000000000200da8 0x0000000000000270	0x0000000000200da8 RW 0x200000
4 DYNAMIC	0x0000000000000db8 0x00000000000001f0	0x0000000000200db8 0x00000000000001f0	0x0000000000200db8 RW 0x8
5 NOTE	0x0000000000000254 0x0000000000000044	0x0000000000000254 0x0000000000000044	0x0000000000000254 R 0x4
GNU_EH_FRAME	0x0000000000000804 0x000000000000003c	0x0000000000000804 0x000000000000003c	0x0000000000000804 R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW 0x10
GNU_RELRO	0x0000000000000da8 0x0000000000000258	0x0000000000200da8 0x0000000000000258	0x0000000000200da8 R 0x1

Section to Segment mapping:

Segment	Sections...
00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got
xt	.fini .rodata .eh_frame_hdr .eh_frame
03	.init_array .fini_array .dynamic .got .data .bss

- 1 PHDR specifies the location and size of the program header table itself, both in the file and in the memory image of the program.
- 2 INTERP specifies location and size of an interpreter for linking runtime library.
- 3 The LOAD directives determinate what parts of the ELF file get mapped into program memory.
- 4 The DYNAMIC directives dynamic linking information.
- 5 The NOTE indicate of auxiliary information.

```
root@slingshot:/home# readelf -l 484b_test
```

Elf file type is DYN (Shared object file)

Entry point 0x5f0

There are 9 program headers, starting at offset 64

Program Headers:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags	Align
PHDR	0x0000000000000040 0x00000000000001f8	0x0000000000000040 0x00000000000001f8	0x0000000000000040 R	 0x8
INTERP	0x0000000000000238 0x000000000000001c	0x0000000000000238 0x000000000000001c	0x0000000000000238 R	 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]				
LOAD	0x0000000000000000 0x0000000000000948	0x0000000000000000 0x0000000000000948	0x0000000000000000 R E	 0x200000
LOAD	0x0000000000000da8 0x0000000000000268	0x0000000000200da8 0x0000000000000270	0x0000000000200da8 RW	 0x200000
DYNAMIC	0x0000000000000db8 0x00000000000001f0	0x0000000000200db8 0x00000000000001f0	0x0000000000200db8 RW	 0x8
NOTE	0x0000000000000254 0x0000000000000044	0x0000000000000254 0x0000000000000044	0x0000000000000254 R	 0x4
1 GNU_EH_FRAME	0x0000000000000804 0x000000000000003c	0x0000000000000804 0x000000000000003c	0x0000000000000804 R	 0x4
2 GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW	 0x10
3 GNU_RELRO	0x0000000000000da8 0x0000000000000258	0x0000000000200da8 0x0000000000000258	0x0000000000200da8 R	 0x1

Section to Segment mapping:

Segment Sections...

00

01 .interp

02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got

xt .fini .rodata .eh_frame_hdr .eh_frame

03 .init_array .fini_array .dynamic .got .data .bss

1 GCC uses this table to find the appropriate handler for an exception.

2 whether we need an executable stack; permission of the stack in memory.

3 which part of the memory should be read-only after applying dynamic relocations

PHT Inspecting...

LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000	
	0x00000000000000948	0x00000000000000948	<u>RE</u>	0x200000
LOAD	0x0000000000000da8	0x00000000000200da8	0x00000000000200da8	
	0x0000000000000268	0x0000000000000270	<u>RW</u>	0x200000

❑ Load Segment appear twice. Why?

First LOAD has read and execute permission. Therefore, this segment running *text* segment. Because only text segment contain read-only instruction with read-only data section (go to next page for more detailed).

Second LOAD has read and write permission. So this is a *data* segment. Notice that this segment can not executable.

PHT Inspecting...

- All segments contains sections.

Segment Sections...

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text
.rodata	.eh_frame_hdr .eh_frame
03	.init_array .fini_array .dynamic .got .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	.eh_frame_hdr
07	
08	.init_array .fini_array .dynamic .got

➔ Index of Segments. In other word, The first number is the index of a program header in program header table, and the remaining text is the list of all sections within a segment.

Look at the number 2 index, this section belonging to First LOAD segment. So, first LOAD segment contains the number 2 index sections.

Section Header Table

- The section headers define all the sections within an ELF-file. In the section header the data is linked and relocated. The section header table describes zero or more sections that are followed by data which are referred to by entries from the program header table, or section header table.(6)

Some Sections...

Sections:

- .text -> contains executable code, which will be packed into a segment with read and execute access rights. Which is only loaded once, as the contents will not change.
- .rodata -> Initialized data with read access rights only
- .data -> Initialized data with read/write access rights
- .bss -> uninitialized data with read/write access rights(6)


```
root@slingshot:/home# readelf -W -S 484b_test
```

There are 29 section headers, starting at offset 0x1990:

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	0000000000000238	000238	00001c	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	0000000000000254	000254	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	0000000000000274	000274	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	0000000000000298	000298	00001c	00	A	5	0	8
[5]	.dynsym	DYNSYM	00000000000002b8	0002b8	0000d8	18	A	6	1	8
[6]	.dynstr	STRTAB	0000000000000390	000390	0000a4	00	A	0	0	1
[7]	.gnu.version	VERSYM	0000000000000434	000434	000012	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	0000000000000448	000448	000030	00	A	6	1	8
[9]	.rela.dyn	RELA	0000000000000478	000478	0000c0	18	A	5	0	8
[10]	.rela.plt	RELA	0000000000000538	000538	000048	18	AI	5	22	8
[11]	.init	PROGBITS	0000000000000580	000580	000017	00	AX	0	0	4
[12]	.plt	PROGBITS	00000000000005a0	0005a0	000040	10	AX	0	0	16
[13]	.plt.got	PROGBITS	00000000000005e0	0005e0	000008	08	AX	0	0	8
[14]	.text	PROGBITS	00000000000005f0	0005f0	000202	00	AX	0	0	16
[15]	.fini	PROGBITS	00000000000007f4	0007f4	000009	00	AX	0	0	4
[16]	.rodata	PROGBITS	0000000000000800	000800	000004	04	AM	0	0	4

SHT Inspecting ...

Section Headers:

[Nr]	Name
[0]	
[1]	.interp

Index = 1

Type
NULL
PROGBITS

PROGBITS = which means this section is part of the program.

Address	Off	Size	ES	Flg	Lk	Inf	Al
0000000000000000	000000	000000	00		0	0	0
0000000000000238	000238	00001c	00	A	0	0	1

Offset

Flag = are A (Allocatable) which means this section consumes memory at runtime.

Address = means the program is loaded at this virtual memory address at runtime.

EntSize = is 0, which means this section does not have any fixed-size entry.

Alignment

Link and Info are 0 and 0 means this section links to no section or entry in any table

Reference

- 1 - Ubuntu linux -> /usr/include/elf.h
- 2 - Espen, Amar & Abdi, “Automated dynamic malware analysis of ELF-files”
- 3 - <https://www.linuxjournal.com/article/6463>
- 4 - <https://refspecs.linuxfoundation.org/elf/elf.pdf>
- 5 - https://web.stanford.edu/~ouster/cs111-spring21/all_lectures/
- 6 - A. Dennis, "Practical binary analysis : build your own Linux tools for binary", No Starch Press, 2019.
- 7 - https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

To be Continued ...