Alee's Stories ELF's Story Part2: ELF's Structure: ELF Header

aleeamini.com/elfs-story-part2-elfs-structure-elf-header/

ELF’s Story Part2: ELF’s Structure: ELF Header

1-Introduction

In the previous part of this story, we learned about compilation, linking, and how a linker works. In this part, I would like to discuss the ELF (Executable and Linkable Format) structure and the elements that make up an ELF file.

An ELF file is not just a binary code that executes on a CPU. It has a specific structure that serves as an identifier, providing information about the file’s properties such as size, type, architecture, data, and more. When an executable file is run on an operating system, the OS reads the file’s structure to determine how to run it.
For example, the OS reads the structure to locate the executable codes and then begins running them.
To understand how an OS can load and run an ELF file, we need to learn about the ELF file structure.
Before we dive into the details, let’s define what an ELF file is.
The ELF file is a binary format used for executable files, shared objects (*.so), and core dump files. In this series, we will be discussing 64-bit ELF files, but the structure of 32-bit and 64-bit ELF files is very similar.
An ELF file consists of four primary components: an ELF header, some optional program or segment headers, several sections, and a series of optional section headers. The program and section headers are tables for every segment or section, meaning that if we have four sections, we have four section headers.
Now that we have gone over the components, let’s move on to understanding the details of the ELF file structure.

2-ELF Header

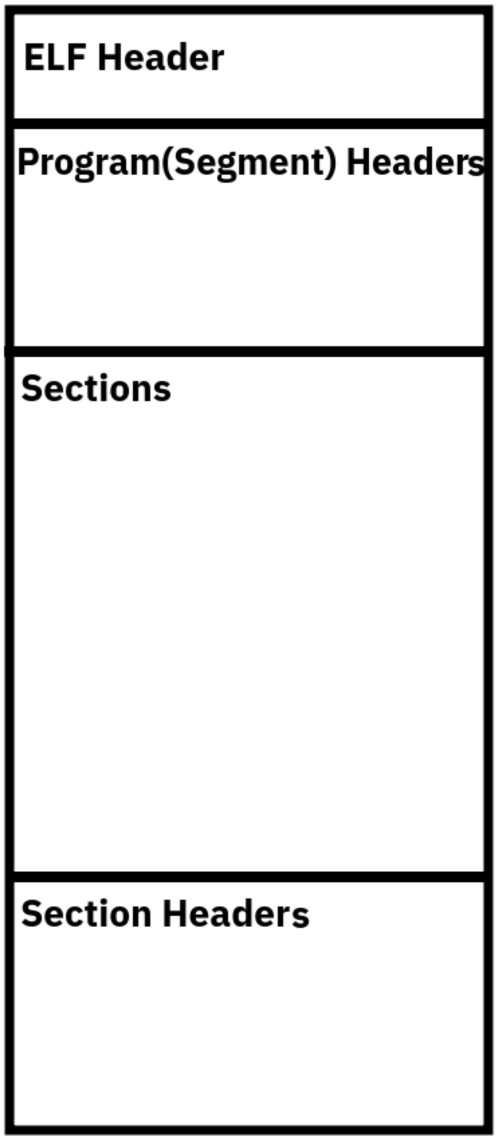
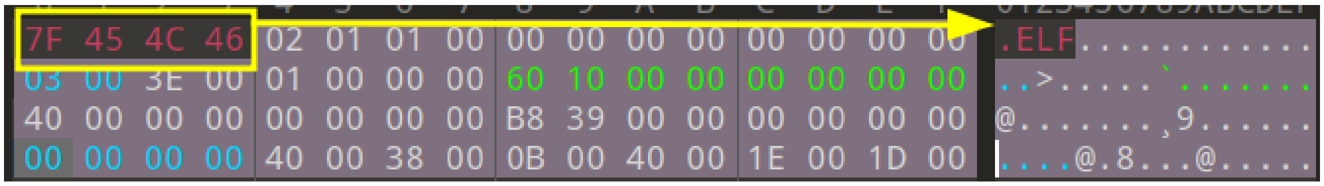


Figure 2-1: ELF File Components

The first component of an ELF file is the ELF header, which is located at the beginning of the file. This header contains important information about the ELF file, including its type, architecture, and the location of other parts within the file. Essentially, the header provides a basic structure of bytes that serves as an introduction to the ELF file. In this text, I will describe all parts of the ELF header in further detail.

2-1 Identifier

The first 16 bytes of this header are identifiers. First 4 bytes of this array are the magic value of the ELF file. The first byte is 0x7f and the rest 3 bytes are E, L, and F letters.
[0x7f,’E’,’L’,’F’]



```
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
03 00 3E 00 01 00 00 00 60 10 00 00 00 00 00 00 ..>.....`.......
40 00 00 00 00 00 00 00 B8 39 00 00 00 00 00 00 @.......¸9......
00 00 00 00 40 00 38 00 0B 00 40 00 1E 00 1D 00 ....@.8...@.....
```

Figure 2-2: ELF Identifier bytes

As you can see the magic number is the first thing that we see in an ELF file. This magic value is at the beginning of the file due to tools that scan files like the ‘file’ tool in Linux can detect this file as an ELF.
The rest bytes of this 16-byte array are some other values that describe the ELF file better for us.

EI_CLASS: The 5th byte of this array is the EI_CLASS byte that specifies whether this binary file is 32-bit or 64-bit. The value of 1 is for 32-bits and 2 for 64-bits.

EI_DATA: The 6th byte of this array is EI_DATA. This byte specifies the order of multi-byte values like numbers. In other words, this byte specifies the endian of the binary file. If it is Little-Endian set it to 1 and if be Big-Endian, set it to 2.

EI_VERSION: The 7th byte of this array is EI_VERSION. This byte specifies the version of the ELF that is used in this file. The only valid version of the ELF structure until now is 1 and there is no other version for the ELF structure. So it should be 1 always.

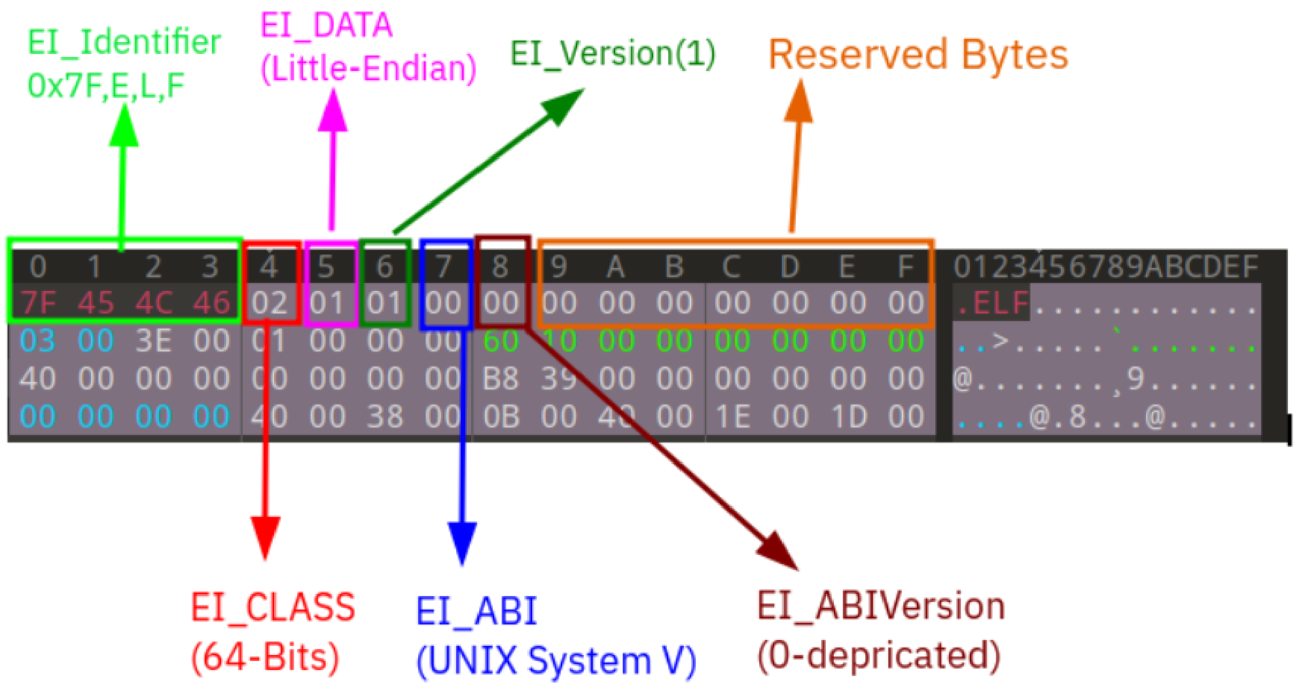
EI_OSABI: The 8th byte of this array is EI_OSABI. This byte specifies the Application Binary Interface. In other words, this value is the OS that this binary file could run on. In regular ELF files, this byte is 0x00 which means is for the “UNIX System V” version of Unix-based OSes. You can see other values for this byte:

Value	ABI
0x00	**System V**
0x01	HP-UX
0x02	NetBSD
0x03	Linux
0x04	GNU Hurd
0x06	Solaris
0x07	AIX (Monterey)
0x08	IRIX
0x09	FreeBSD
0x0A	Tru64
0x0B	Novell Modesto
0x0C	OpenBSD
0x0D	OpenVMS
0x0E	NonStop Kernel
0x0F	AROS
0x10	FenixOS
0x11	Nuxi CloudABI
0x12	Stratus Technologies OpenVOS

Table 2-1: OS ABI Values
Ref: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

EI_ABIVERSION: The **9th** byte of this array is EI_ABIVERSION. This byte specifies the Application Binary Interface version. In the Linux version after 2.5, this byte always has been 0 and useless.

Reserved_Bytes: The other 7 bytes of this array are reserved bytes. They haven't any used until now and maybe later they will be used for a purpose.



```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00   .ELF............
03 00 3E 00 01 00 00 00 60 10 00 00 00 00 00 00   ..>.....`.......
40 00 00 00 00 00 00 00 B8 39 00 00 00 00 00 00   @.......¸9......
00 00 00 00 40 00 38 00 0B 00 40 00 1E 00 1D 00   ....@.8...@.....
```

Figure 2-3:ELF Header Identifier

2-2 Type

After the **16-Byte** Identifier fields, we have a 2-Byte value that specifies the type of ELF binary. These 2-Byte make some states that specify what type is this binary file. As I said before, we have multiple types of ELF files. For example, one of them is Executable ELFs and another one is **Dynamic libraries or Shared objects**. This value specifies the type of the binary file.

Value	Type	Meaning
0x00	ET_NONE	Unknown.
0x01	ET_REL	Relocatable file.
0x02	ET_EXEC	Executable file.
0x03	ET_DYN	Shared object.
0x04	ET_CORE	Core file. (Dump files)
0xFE00	ET_LOOS	Reserved inclusive range. Operating system specific.
0xFEFF	ET_HIOS	
0xFF00	ET_LOPROC	Reserved inclusive range. Processor specific.
0xFFFF	ET_HIPROC	

Table 2-2: e_type values
Ref: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

As you can see, for example, if the value of Type is 0x2, it means that the file is executable, or if it is 0x01, the file is a relocatable file that was talked about in the previous part.

2-3 Machine

The next field is Machine type. It is a 2-Byte field that indicates the machine type and instruction sets of the binary file. But what does this mean? Instruction Set, or ISA, is the architecture and assembly language that is used by the CPU to execute codes. Every CPU vendor has its own architecture and instruction set. For example, the Intel assembly language is different from the ARM assembly. Instructions, call flow, and so on are different. For example in our case, this value is 0x3E which means our ELF binary file instruction set is X86-64 which refers to **64-bit** architecture, because I use from Intel CPU and I installed a **64-bit** Linux OS.

Value	ISA
Value	**ISA**
0x00	No specific instruction set
0x01	AT&T WE 32100

Value	ISA
0x02	SPARC
0x03	x86
0x04	Motorola 68000 (M68k)
0x05	Motorola 88000 (M88k)
0x06	Intel MCU
0x07	Intel 80860
0x08	MIPS
0x09	IBM System/370

Showing 1 to 10 of 70 entries

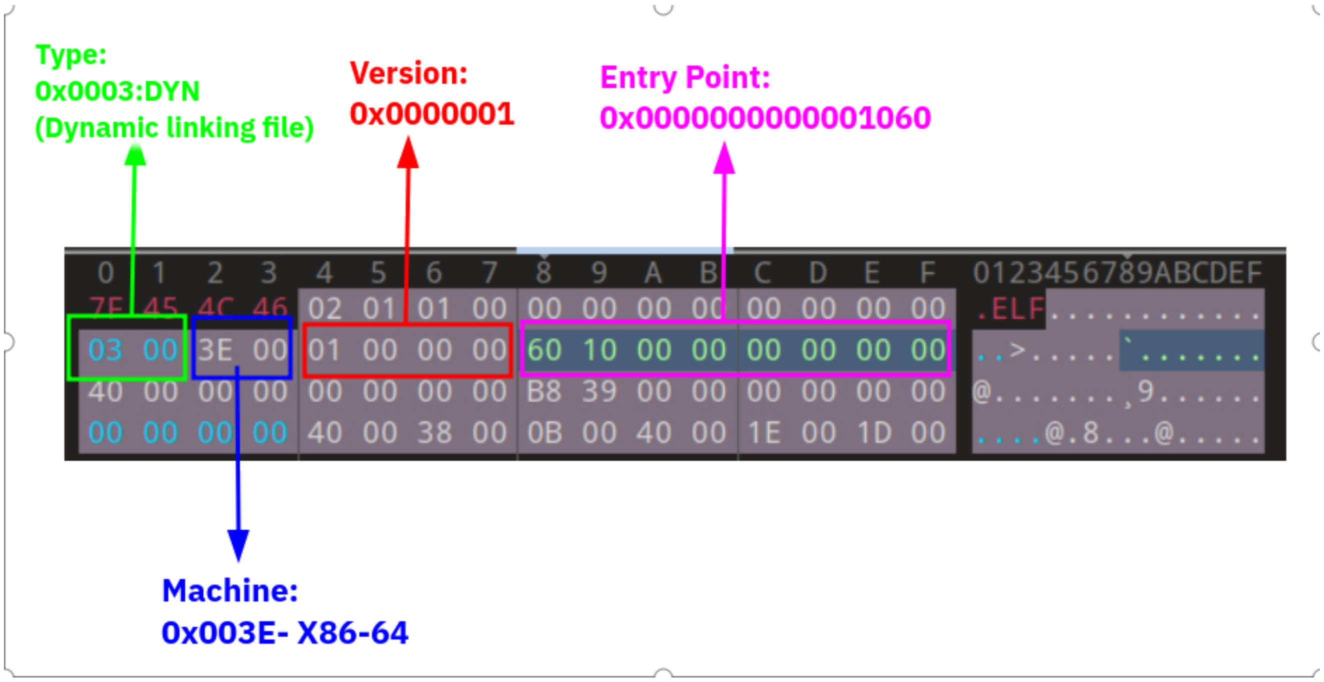
PreviousNext
In the above table, you can see all machine types. In the table are 70 types of machines.

2-4 Version

The next field is the version that is a **4-Byte** value. This field serves the same role as the **EI_VERSION** byte in the Identifier array. Specifically, it indicates the version of the ELF specification that was used when creating the binary. The only true value for this field is 1 (**EV_CURRENT**).

2-5 Entry

The next field is important. This **8-Byte**(**4-Byte** for **32-bit** binaries) value, indicates the virtual address of the Entry Point of the ELF binary. The entry point is the address where the execution should start from there. For example, if the entry point of our binary is 0x400000, the interpreter or dynamic loader (ld-Linux that I talked about in the previous part), transfers the execution to this address after it finishes the loading process and the binary execution starts from here.
I’ll talk about dynamic loading in future parts. This value is in little-endian format.



```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00   .ELF............
03 00 3E 00 01 00 00 00 60 10 00 00 00 00 00 00   ..>.....`.......
40 00 00 00 00 00 00 00 B8 39 00 00 00 00 00 00   @.......¸9......
00 00 00 00 40 00 38 00 0B 00 40 00 1E 00 1D 00   ....@.8...@.....
```

Figure 2-4: ELF Header fields

You can see the visual schema of the ELF header.

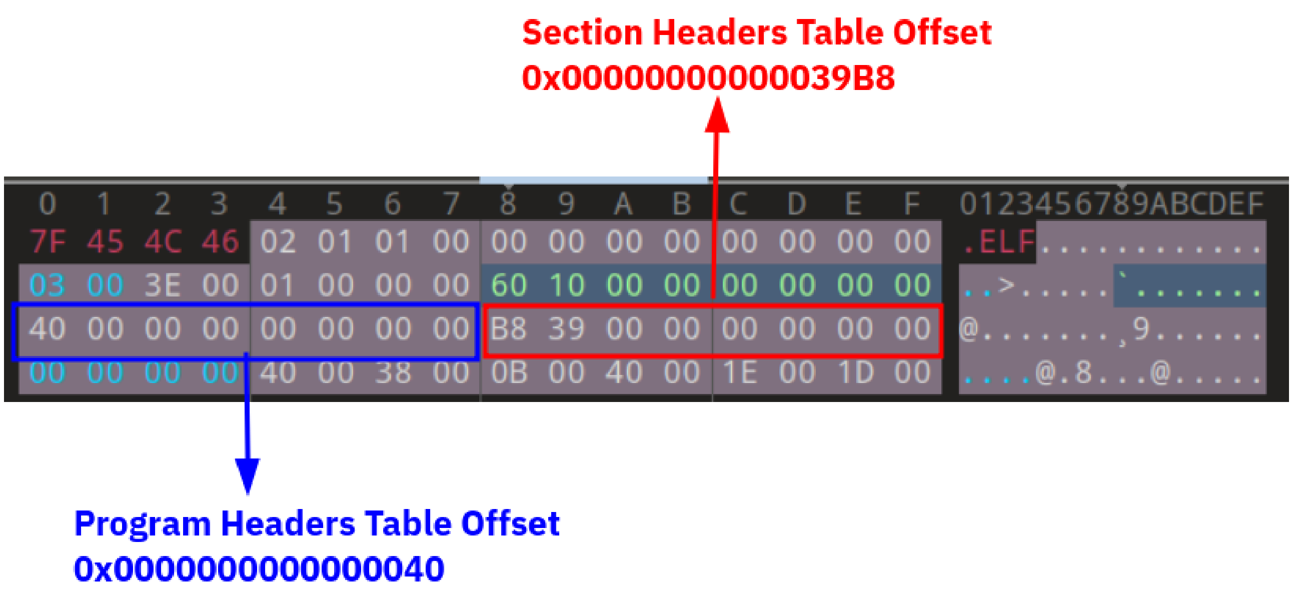
2-6 PhOffset (Program Headers Offset)

This **8-Byte** (4-Byte in 32-bit binaries) field indicates the start offset of the Program Header of the ELF binary. As you know, the Program Headers table is part of an ELF file. However, an ELF file may present in any size.
So we can't guess where is the Program Headers table. Due to this, we need an indicator that specifies it for us.
This field is an offset, so its value means the count of bytes from the start of the ELF file. For example, if the value of PhOffset is 0x0000000000000040, it means that after 0x40 bytes from the start of the file, we reach the beginning of the Program Headers table. This value is in little-endian format.

2-7 ShOffset (Section Headers Offset)

This 8-Byte (4-Byte in 32-bit binaries) field indicates the start offset of the Section Headers table of the ELF binary. This field is like the PhOffset field. This value is in little-endian format.
I should tell you that, the Program Header and Section Headers table
don't need to be located at any particular offset in the ELF file.
The only part that should be at a fixed location is the ELF Header, which is always at the beginning of the ELF file.
If you back and see Figure 1-1, you see that the Program Header table is located after the ELF Header, and the Section Headers table is located at the end of the file. But this is not always in this way, because all components of the ELF file can be located at any location,

except the ELF Header that is located at the beginning of the file. The reason is clear, we can find all components of the ELF file, so we need an address table at a fixed location, to access it easily and find addresses of other parts of the ELF file.
So I can tell that The most important structure of an ELF file is the ELF Header. Because with it, we can find all other parts. The ELF Header is the map of an ELF file.



```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00   .ELF............
03 00 3E 00 01 00 00 00 60 10 00 00 00 00 00 00   ..>.....`.......
40 00 00 00 00 00 00 00 B8 39 00 00 00 00 00 00   @.......¸9......
00 00 00 00 40 00 38 00 0B 00 40 00 1E 00 1D 00   ....@.8...@.....
```

Figure 2-5: ELF Header PhOffset and ShOffset

2-8 Flags

The next field of the ELF Header is flagged. This 4-Byte value specifies some flags for the processor that wants to run the ELF. These flags are CPU specifics, so their value may differ from one CPU to another. For example, for those binaries that are compiled for running in ARM devices, the compiler may set some flags in this field to tell the ARM CPU, how to run the binary. For X86 binaries, this flag is typically set to zero.

2-9 EHSize (ELF Header Size)

The next 2-Bytes field specifies the size of the ELF Header in bytes. But what do we need the size of the ELF Header? This is because when we want to parse the ELF file, we should know how many bytes we should pass to arrive at the next part of the file. This value is usually 64-Bytes for 64-bit and 52-Bytes for 32-bit format.

Let’s ask a question: We want to parse the Program Headers table or Section Headers table. Until this moment, we have the offset of these parts with PhOffset and ShOffset fields. So we

can locate the beginning of them. For example, we arrive at the beginning of the Program Headers table, at the first Header. But how to traverse them and find out how many bytes should be read to arrive at the next Header?
The answer to the question is in the following fields.

2-10 PhEntrySize(Program Headers Entry Size)

This is a 2-Byte field that specifies the size of the one entry of the Program Headers Table. This size is the same in all other entries of the Program Headers table. So we find out the size of every Header.
Now we can traverse Headers in the Program Headers table. But how we can find out how many Headers we should read until the end of the Program Headers table? The answer is in the next field.

2-11 PhNum(Program Headers Entries Numbers)

This 2-Bytes field specifies the number of entries (headers) that exist in the Program Headers table.
So now, with several entries and the size of one of them, we can calculate how many bytes should be read until arrive at the end of the Program Headers table.
Let’s have an example:
The PhEntrySize is 0x0038 (56 in decimal). So every Header in the Program Headers table has a size of 56 bytes. Also, the PhNum is 0x000B(11 in decimal). So we have 11 Headers in the Program Headers table. So the size of the whole Program Headers table is:
Program Headers Table = PhEntrySize * PhNum –> 0x0038 * 0x000B = 0x268(616)bytes.

2-12 ShEntrySize(Section Headers Entry Size)

This is a 2-Byte field that specifies the size of one entry of the Section Headers Table. This size is the same in all other entries of the Section Headers table. The same as PhEntrySize.

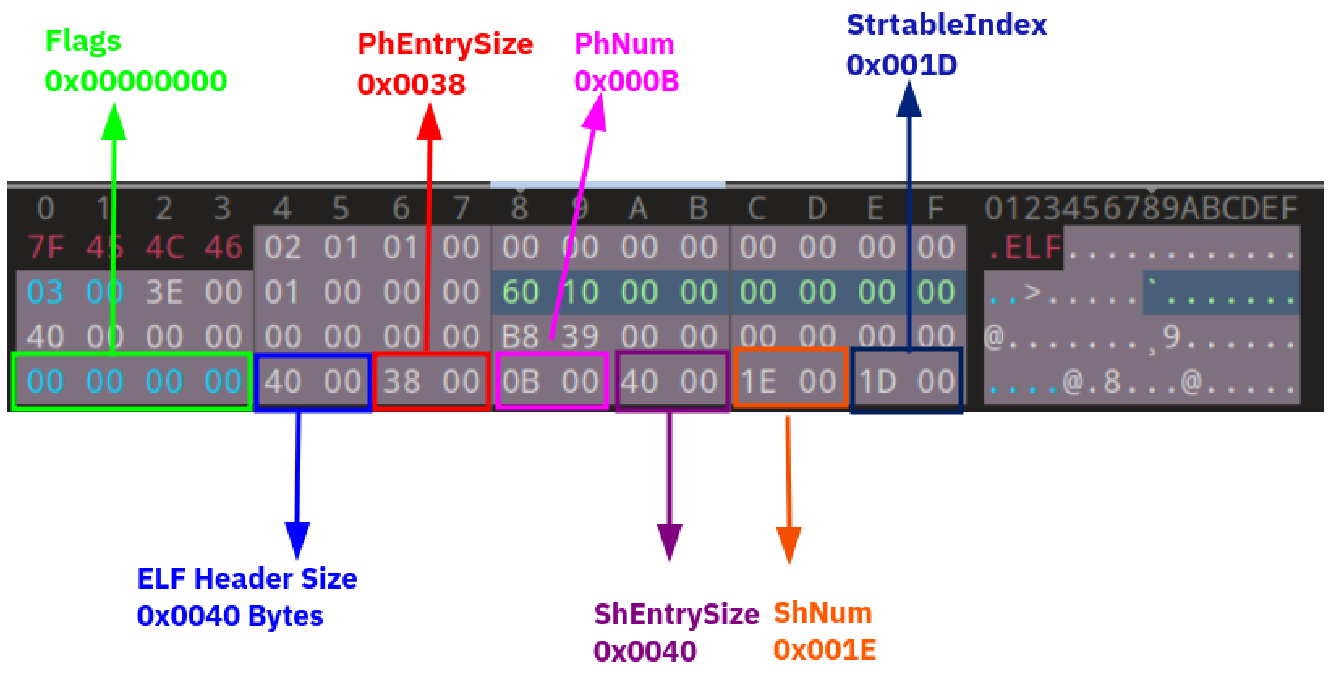
2-13 ShNum(Section Headers Entry Numbers)

This 2-Bytes field specifies the number of entries(headers) that exist in the Section Headers table. Precisely the same as PhNum.

2-14 Shstrndx

In the Sections, we have a table, called **shstrtab**. This is a dedicated section that contains a table of null-terminated ASCII strings, which store the names of all the sections in the binary. The Section Headers String Table field is a 2-Bytes field that contains the index of the

shstrtab table in the Section Headers table. This is needed when some tools want to parse the Section Headers table and need the names of sections. I’ll talk about this table in the future.



```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00  .ELF............
03 00 3E 00 01 00 00 00 60 10 00 00 00 00 00 00  ..>.....`.......
40 00 00 00 00 00 00 00 B8 39 00 00 00 00 00 00  @.......¸9......
00 00 00 00 40 00 38 00 0B 00 40 00 1E 00 1D 00  ....@.8...@.....
```

Figure 2-6: ELF Header in raw

You can see data of the ELF Header of an ELF file with the readelf tool in Linux. This tool is an ELF parser that can parse the ELF and interpret it for you.

```
$ readelf -h main.out
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: 0x1060
Start of program headers: 64 (bytes into file)
Start of section headers: 14776 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 11
Size of section headers: 64 (bytes)
Number of section headers: 30
Section header string table index: 29
```

As you can see the results are clear to us. Now you can match all these values with their binary values.

3-conclusion

Now that you're familiar with the format and contents of the ELF header and the basics of the ELF file.
In this section you learned how to identify an ELF file by parsing the ELF header. For exercise you can write a code that reads an ELF file and then parse the ELF header. Now you can do it on yourself but maybe you will need to use from the elf.h header in /usr/include directory of your Linux.
In the next part of this story, talk about Section headers which are most important part of an ELF file.
let's move on to the next part.

Next Part: ELF's Story Part3_ELF's Structure_Section Headers
Prev Part: ELF's Story Part3_ELF's Structure_

References:

https://eli.thegreenplace.net
https://docs.oracle.com/cd/E23824_01/html/819-0690/
The Practical Binary Analysis Book