ELF's Story Part3: ELF's Structure: ELF Section Headers

aleeamini.com/elfs-story-part3-elfs-structure-elf-section-headers/

ELF’s Story Part3: ELF’s Structure: ELF Section Headers

1-Introduction

Are you ready to delve deeper into the world of ELF files? In my previous part, I discussed the critical role of the ELF header in locating different parts of the file.
Today, I want to share with you some exciting information about the ELF Section Headers. These headers serve as descriptors for various sections of the file, providing valuable insights into their properties.
By learning how to locate and retrieve information from these headers, you can gain a better understanding of the ELF file structure and how it works. So, are you ready to take the next step in your ELF file journey? Let’s dive in together and explore the ELF Section Headers!

2-What is ELF Section Headers

In every binary file, we have two types of contents: **code** and **data**. These contents rest in the binary file in a way that tools like Linker and Loader can load them and use them in the linking time and also in the run time. All this content(codes and data) rests in the ELF file, in some chunks that are named “Section” in the ELF glossary.
So we can imagine that all we have in an ELF file is a series of “Sections”.
These sections do not have any special and predefined structure. the structure of every section depends on its contents.
Sometimes a Section doesn’t have any special structure and is just a series of bytes of data or codes. So we need a thing that can describe a section and identify it for others. So “Section Header” is what we need.

The Section Header is a table that describes a section and denotes the properties of the section.
The Section Headers table contains the Section Headers for all Sections of an ELF binary.

But is important to note that sections are mainly used during the linking phase. This means that they serve as a reference point during the linking process. Section headers play a crucial role in the linking phase of executable files. They provide the linker with important information about the linking process. However, not all sections are necessary during runtime, and as a result, the dynamic loader does not load them into memory when running the executable file.
I'll talk about dynamic loading in the next parts.

Due to that sections are used to provide a view for the linker, the section header table is an optional part in the ELF format. ELF files that don't need linking aren't required to have a section header table. If no second-tion header table is present, the **e_shoff** field in the executable header is set to zero.

OK, every section header has some fields that are to describe a section. These fields are as described below:

2-1 sh_name (Section Name)

This value is a 4-Bytes number that indicates the index of a string in the Sections Headers String table, that is the section name.
As before said, we have a special section in an ELF file which is named section header string table or **.shstrtab**. All the names of sections are saved in it. This section contains some NULL-terminated strings that everyone is for a section.

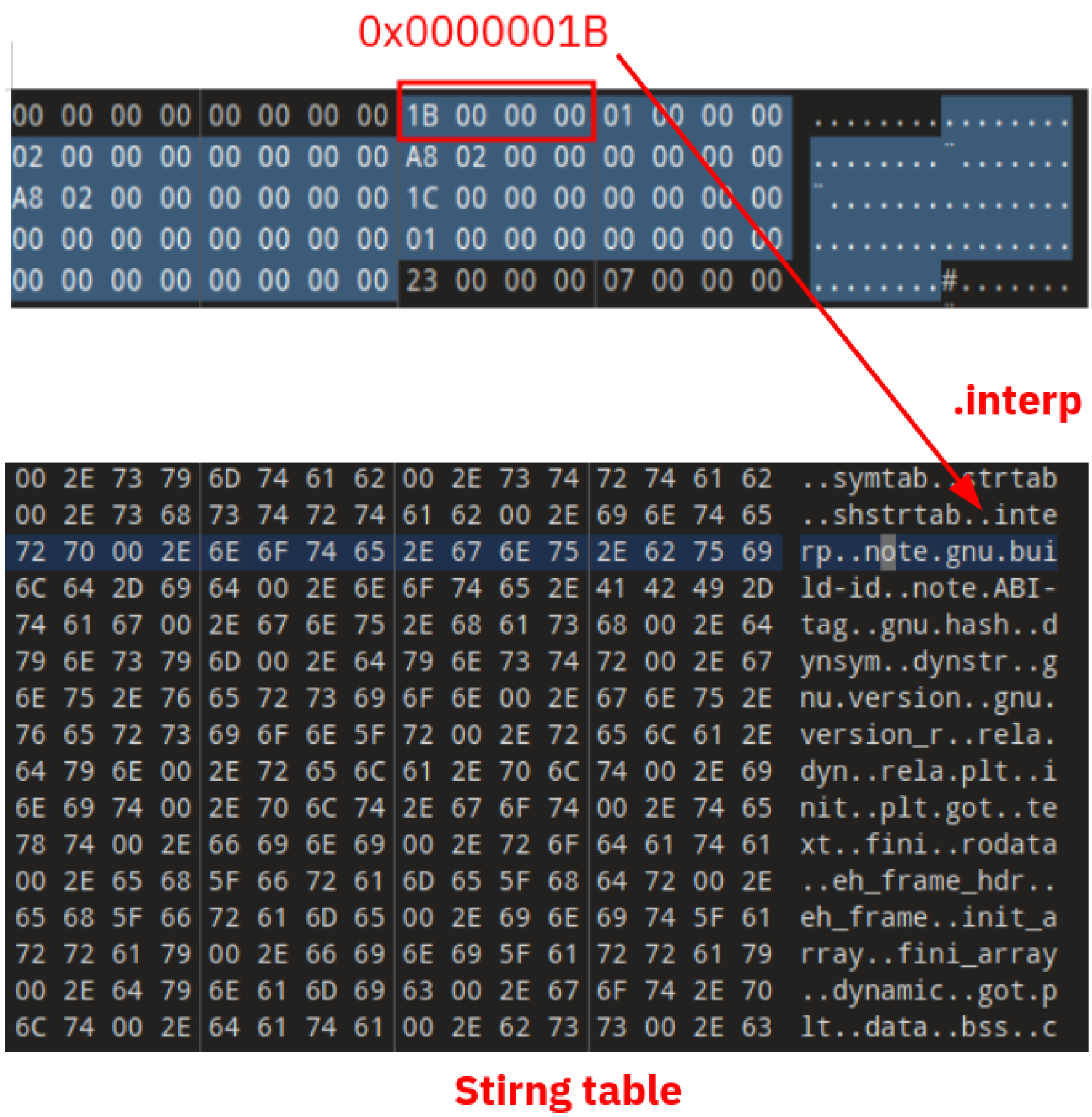

Figure 2-1: The index of section names in the string table

As you see in the above image, in a section header, the value of sh_name is 0x1B or 27. So in the String table, at the 27th index, we can find the name of this section.

2-2 sh_type (Section Type):

This is a **4-Bytes** value that indicates the type of the section. Every section in an ELF file has a special type. This value is useful for the linker at linking time, to detect those sections that are for relocation purposes.

Value	Name	Meaning
0x0	SHT_NULL	Section header table entry unused
0x1	SHT_PROGBITS	Program data
0x2	SHT_SYMTAB	Symbol table
0x3	SHT_STRTAB	String table
0x4	SHT_RELA	Relocation entries with addends
0x5	SHT_HASH	Symbol hash table
0x6	SHT_DYNAMIC	Dynamic linking information
0x7	SHT_NOTE	Notes (Some additional information about the binary)
0x8	SHT_NOBITS	Program space with no data (bss)
0x9	SHT_REL	Relocation entries, no addends
0x0A	SHT_SHLIB	Reserved
0x0B	SHT_DYNSYM	Dynamic linker symbol table
0x0E	SHT_INIT_ARRAY	Array of constructors
0x0F	SHT_FINI_ARRAY	Array of destructors
0x10	SHT_PREINIT_ARRAY	Array of pre-constructors
0x11	SHT_GROUP	Section group
0x12	SHT_SYMTAB_SHNDX	Extended section indices
0x13	SHT_NUM	Number of defined types.
0x60000000	SHT_LOOS	Start OS-specific.

Table 2-1:Section Type values
Ref:https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

I discuss some important types.

SHT_NULL: This type indicates that the section is NULL and there is no data. just a NULL section.

SHT_PROGBITS: This type indicates that the section contains program data such as machine instructions or constants. For example, the opcodes of the executable file, are stored in sections of this type.

SHT_SYMTAB: This type indicates that the section is a static symbol table. A section that it's type is Symbol Table, stores the symbols of the executable in itself as a table.

A symbol is a symbolic name and type for a particular address or offset in the executable file. For example, names of functions and variables are saved as symbols in the ELF file.
Tip: The sections with SHT_SYMTAB are those sections that are used in linking time. The linker can use them to locate functions and variable addresses.

SHT_DYNSYM: This type indicates that the section is a dynamic symbol table. A section that its type is a dynamic symbol table, stores the symbols that are needed at runtime of the executable in itself as a table.
Tip: The sections with SHT_DYNSYM are those sections that are used in running time. The dynamic linker (loader) can use them to locate external functions that should resolve.

SHT_STRTAB: This type indicates that the section is a string table. As before said, the .shstrtab section, holds the names of all sections. This section's type is SHT_STRTAB. These sections hold the names of other parts of the ELF file. They involve some NULL-terminated strings.

SHT_RELA and SHT_REL: This type indicates that the section has information about relocation that is used by the linker at the linking phase. These sections are needed just for linking time.

SHT_DYNAMIC: This type indicates that the section contains information needed for dynamic linking at loading time.

SHT_INIT_ARRAY: This type indicates that the section contains the array of addresses of constructor functions. A constructor function is a function that runs before the main function of the executable. I'll talk about it in the next parts.

SHT_FINI_ARRAY: This type indicates that the section contains the array of addresses of destructor functions. A destructor function is a function that runs before the executable ends.

OK, I'll talk about other types in section parts. now let's continue to talk about other values of a section header.

2-3 sh_flags (Section Flags)

This is an 8-Byte value (4-Byte in 32-bit) that indicates some additional information about the section. The most important values of this field are:

SHF_WRITE: This flag indicates that the section is writable at runtime. this means this section will be used at runtime.

SHF_ALLOC: This flag indicates that the contents of the section will load to a memory buffer at running time.

SHF_EXECINSTR: This flag indicates that the contents of the section are some executable instructions. This means the section contains some code and should load at the running time.

0x1	SHF_WRITE	Writable
0x2	SHF_ALLOC	Occupies memory during execution
0x4	SHF_EXECINSTR	Executable
0x10	SHF_MERGE	Might be merged
0x20	SHF_STRINGS	Contains null-terminated strings
0x40	SHF_INFO_LINK	'sh_info' contains SHT index
0x80	SHF_LINK_ORDER	Preserve order after combining
0x100	SHF_OS_NONCONFORMING	The section is member of a group
0x200	SHF_GROUP	The section is excluded unless referenced or allocated (Solaris)
0x400	SHF_TLS	Section hold thread-local data
0x0FF00000	SHF_MASKOS	OS-specific
0xF0000000	SHF_MASKPROC	Processor-specific
0x4000000	SHF_ORDERED	Special ordering requirement (Solaris)
0x8000000	SHF_EXCLUDE	Section is excluded unless referenced or allocated (Solaris)

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

2-4 sh_addr (Section Address)

This 8-Byte (4-Byte in 32-bit) value is the address of the section in virtual memory. This value is valid for those sections that will loaded at runtime. For the sections that don't load at running time, this value is zero.

As I mentioned earlier, sections are utilized during the linking phase rather than at runtime. But here we see a value that indicates the address of the section at running time in memory. what happened? This is for static linker. some parts of sections will load at memory at the running time and in this case, the static linker should know about them to relocate them correctly.

2-5 sh_offset (Section Offset)

This 8-Byte (4-Byte in 32-bit) value is the offset of the section in the ELF file. This field specifies the offset from the beginning of the file to the start of the section.

2-6 sh_size (Section Size)

This 8-Byte (4-Byte in 32-bit) value is the size of the section in bytes.

2-7 sh_link (Section Link)

This 4-Byte value indicates the index number of an associated section. Some sections have a relationship with other sections. For example, sections that are in SHT_SYMTAB, SHT_DYNSYM, or SHT_DYNAMIC types, usually have a linked section that is a string table section that contains the symbolic names for the symbols in question.

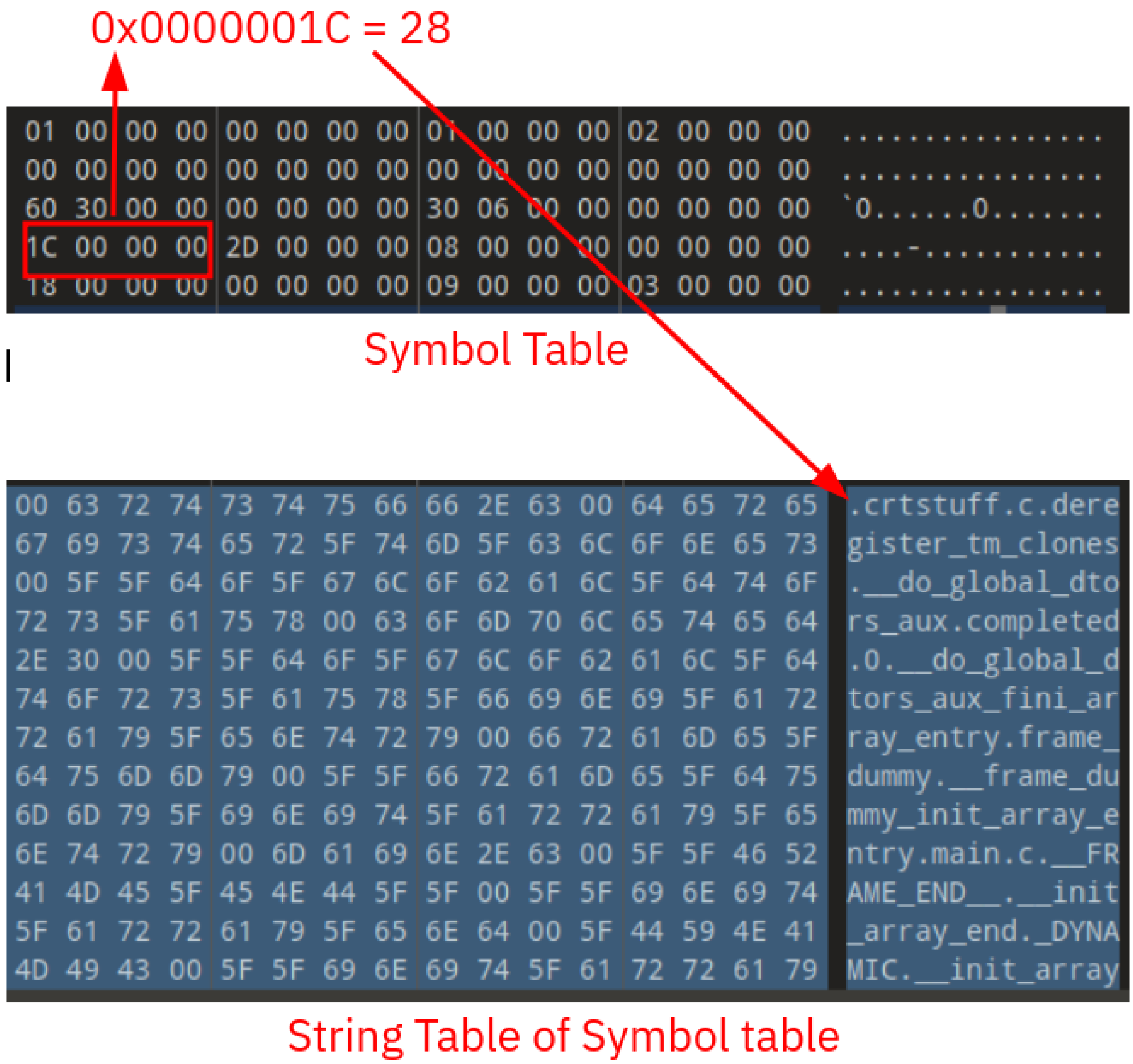

Figure2-2: Section Link

As you can see in the above image, in the symbol table (SH_SYMTAB type) the value of the sh_link is 0x1c (28) which is an index of the 28th section in the ELF file. then we can iterate in section headers and locate the 28th section. The 28th is a string table-type section containing some NULL-terminated strings used in the symbol table section.

2-8 sh_info (Section Information)

This 4-Byte value indicates some more information about the section. This value varies depending on the section type. For example, this value in sections with relocation type is the index of the section that should relocate at linking time.

2-9 sh_addralign (Section Address Align)

This 8-Byte (4-Byte in 32-bit) value is the alignment of the section. Some sections should be mapped with a particular size. So this value indicates the value of alignment. This value must be a power of two. For example, if this field is set to 8, the base address of the section (as chosen by the static linker) must be some multiple of 8. The values 0 and 1 are reserved to indicate no special alignment needs.

2-10 sh_entsize (Section Entry Size)

Some sections, such as symbol tables or relocation tables, contain a table of well-defined data structures (such as Elf64_Sym or Elf64_Rela). For such sections, the sh_entsize field indicates the size in bytes of each entry in the table. When the field is unused, it is set to zero

Now we can take a look at the section headers of an ELF file. You can use from readelf tool in Linux.

```
alee@Debian-Laptop:~/elfs_story/codes$ readelf --section-headers main.out -W
There are 30 section headers, starting at offset 0x39b8:

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        00000000000002a8 0002a8 00001c 00   A  0   0  1
  [ 2] .note.gnu.build-id NOTE            00000000000002c4 0002c4 000024 00   A  0   0  4
  [ 3] .note.ABI-tag     NOTE            00000000000002e8 0002e8 000020 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0000000000000308 000308 000024 00   A  5   0  8
  [ 5] .dynsym           DYNSYM          0000000000000330 000330 0000c0 18   A  6   1  8
  [ 6] .dynstr           STRTAB          00000000000003f0 0003f0 00009d 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          000000000000048e 00048e 000010 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         00000000000004a0 0004a0 000030 00   A  6   1  8
  [ 9] .rela.dyn         RELA            00000000000004d0 0004d0 0000c0 18   A  5   0  8
  [10] .rela.plt         RELA            0000000000000590 000590 000030 18  AI  5  23  8
  [11] .init             PROGBITS        0000000000001000 001000 000017 00  AX  0   0  4
  [12] .plt              PROGBITS        0000000000001020 001020 000030 10  AX  0   0 16
  [13] .plt.got          PROGBITS        0000000000001050 001050 000008 08  AX  0   0  8
  [14] .text             PROGBITS        0000000000001060 001060 000211 00  AX  0   0 16
  [15] .fini             PROGBITS        0000000000001274 001274 000009 00  AX  0   0  4
  [16] .rodata           PROGBITS        0000000000002000 002000 000038 00   A  0   0  8
  [17] .eh_frame_hdr     PROGBITS        0000000000002038 002038 000044 00   A  0   0  4
  [18] .eh_frame         PROGBITS        0000000000002080 002080 000128 00   A  0   0  8
  [19] .init_array       INIT_ARRAY      0000000000003de8 002de8 000008 08  WA  0   0  8
  [20] .fini_array       FINI_ARRAY      0000000000003df0 002df0 000008 08  WA  0   0  8
  [21] .dynamic          DYNAMIC         0000000000003df8 002df8 0001e0 10  WA  6   0  8
  [22] .got              PROGBITS        0000000000003fd8 002fd8 000028 08  WA  0   0  8
  [23] .got.plt          PROGBITS        0000000000004000 003000 000028 08  WA  0   0  8
  [24] .data             PROGBITS        0000000000004028 003028 000010 00  WA  0   0  8
  [25] .bss              NOBITS          0000000000004038 003038 000008 00  WA  0   0  1
  [26] .comment          PROGBITS        0000000000000000 003038 000027 01  MS  0   0  1
  [27] .symtab           SYMTAB          0000000000000000 003060 000630 18     28  45  8
  [28] .strtab           STRTAB          0000000000000000 003690 000220 00      0   0  1
  [29] .shstrtab         STRTAB          0000000000000000 0038b0 000107 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),
  l (large), p (processor specific)
```

Figure 2-3: readelf – section headers

In the above image, we can see the output of the readelf. This is the section headers of an ELF file. As you can see there are 30 (0-29) section headers. Also, the first section header is for a NULL section with index 0. After this section header, other sections are listed. As I said before, a section header is an identifier for a section. So for example section header 14 describes a section named ".text", for us.

We find out from this section header that the name of the section is ".text" and its type is "PROGBITS", which means the section contains program codes. Also, there is its Address, offset, and size. If you look at flags of the ".text" section, flags A and X are present. They indicate that the section should load in memory at running time and also its contents are executable (X). So we find out the ".text" section will load at running time. There is no link section and information for this section but there is an alignment with 16 value. This means the target memory address that the loader allocates for this section, must be a multiple of 16 in the virtual memory of the process.

Example

In the next part of this story, I will discuss the sections in more detail. However, I understand that this part can be complex and confusing. To help with this, I recommend a practical exercise to better understand the sections and section headers.

OK let's do it.
Imagine we want to locate section headers in an ELF file and then locate a section with the name ".interp".
For this exercise, we need a Hex Editor. I use 010-editor. It has a great UI/UX and it is free for 30 days.
I compiled the code from the previous part and now we have a file named main.out. I open it in the hex editor. I don't want to use any script for this exercise. All things are by hand. From the previous section of this part, we learned what is ELF header and what is its structure. If you remember, in the ELF header we have a value named **ShOffset** that indicates the offset of beginning the section headers table. this value is located at the 40th index of the beginning of the file and its size is 8-Byte in 64-bit. So we can easily find it. Also, we know that this is a little-endian value.

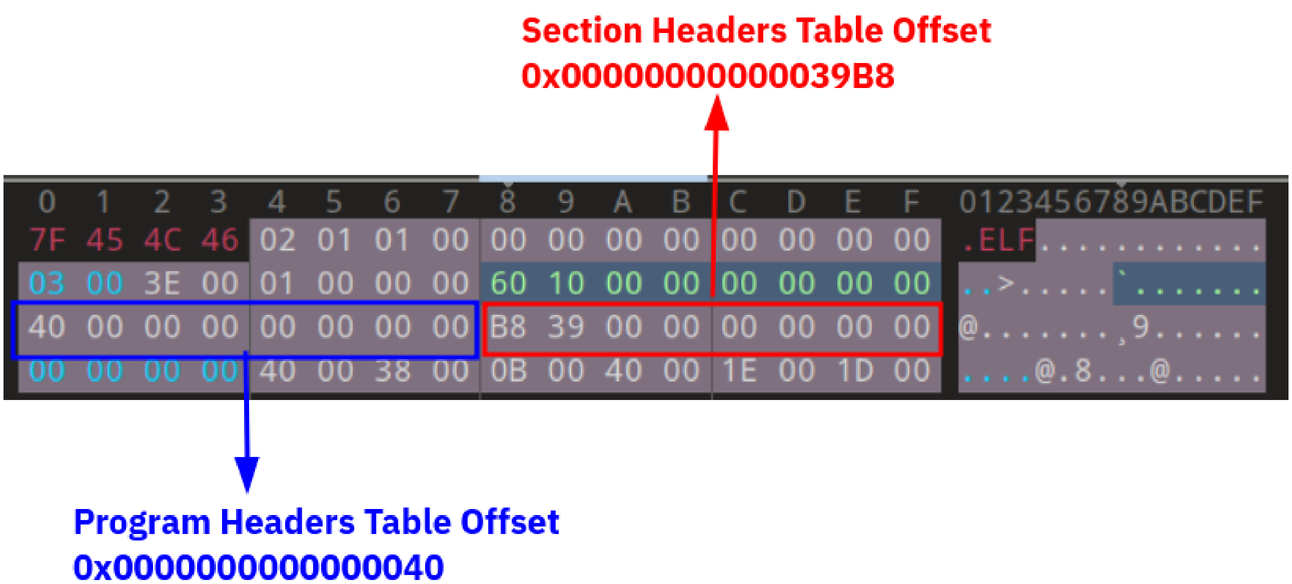

Figure 2-4: Locating Section header table offset

As you see the value of the Section headers table offset is: "**0x00000000000039B8**".
Now we need to go to the 0x39B8th byte of the file to arrive at the beginning of the Section headers table. In the 010-editor you can go to any offset of the file by pressing ctrl+G. I press it and then enter the value 0x39B8 in it and press Enter. Now We are at the beginning of the Section headers table.

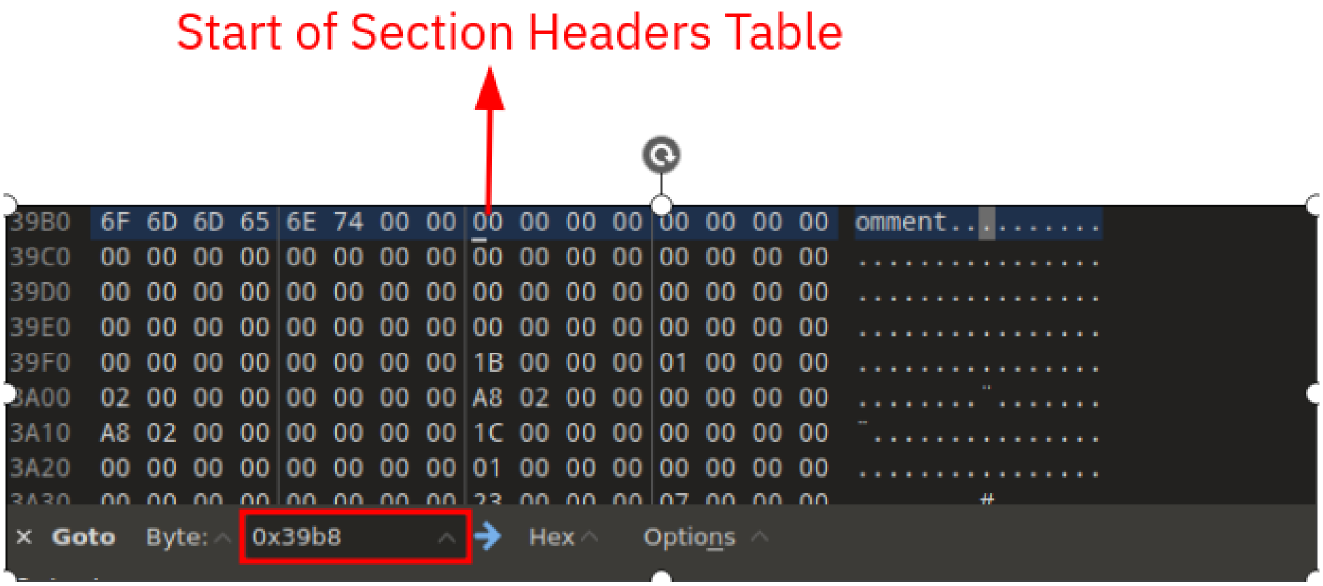

Figure 2-5: Beginning of the Section headers table

But we have another value in the ELF header that could help us find the size of the section headers table. ShEntrySize tells us the size of every section header entry and the ShNum value of the ELF header is the number of all exsiting sections. So we can calculate the whole size of the section headers table.

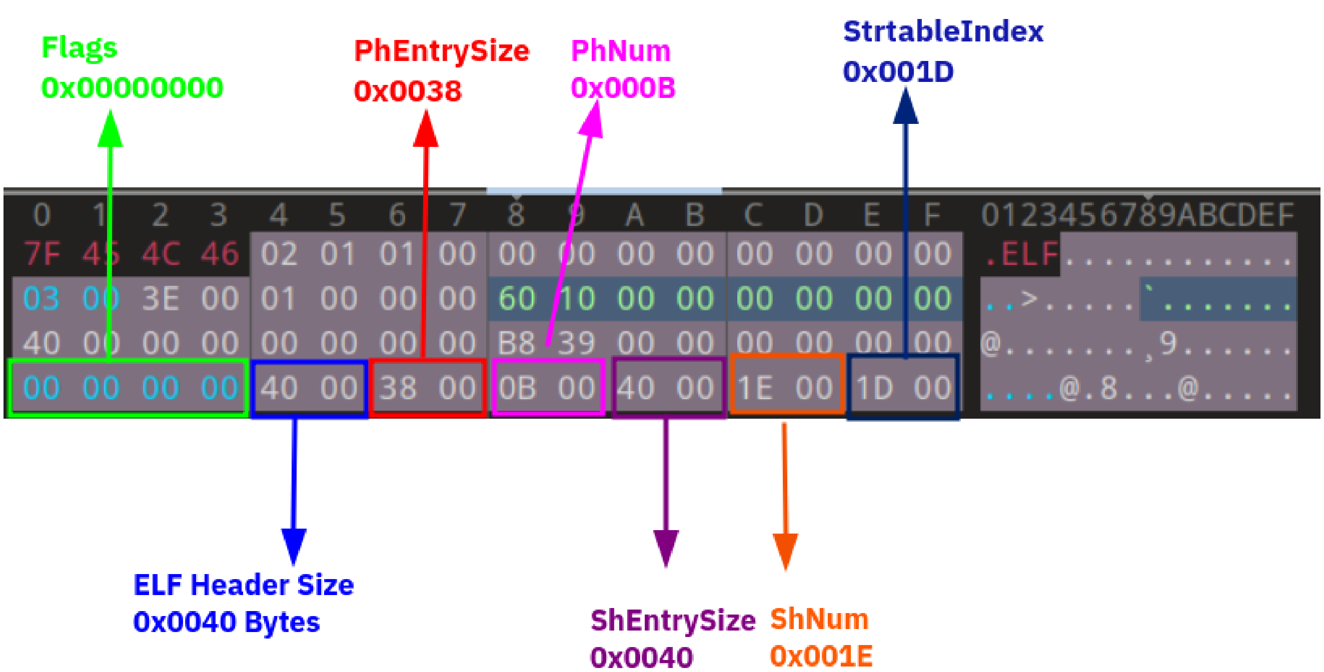

Figure 2-6: Section header entry Size and Numbers

The Whole size of the section headers table: 0x1E*0x40=0x780 bytes. So we can select 0x780 bytes from the beginning of the section headers table to select all section headers table. You can do it by pressing Ctrl+Shift+A to select a range in 010-editor or select 1920 bytes by selecting them by mouse.

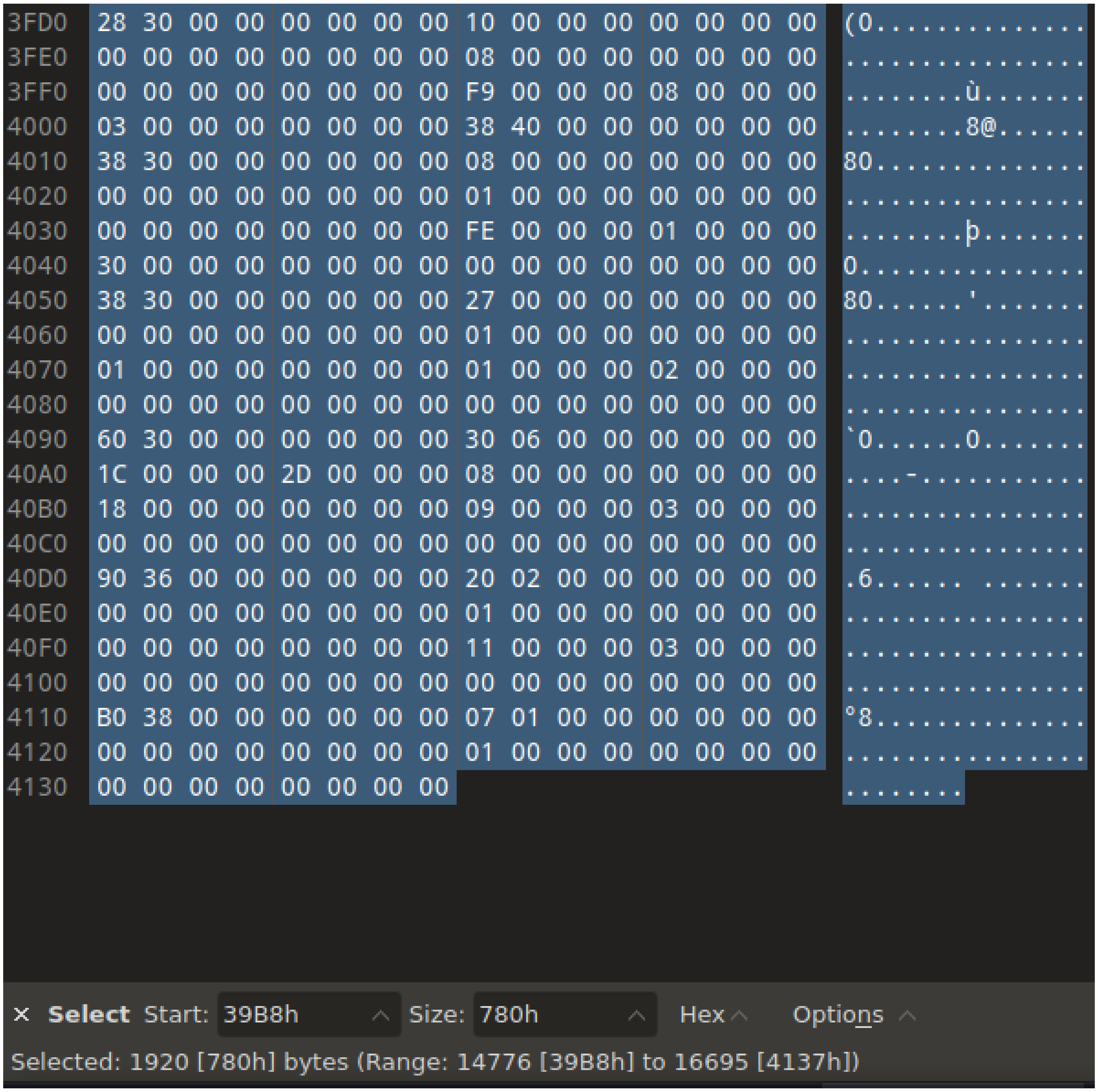

Figure 2-7: The whole section headers table

As you see in the above image, I selected 0x780 bytes from offset 0x39B8 of the beginning of the section headers table.

I reached the end of the file. So we can find out that the section headers table is located at the end of this ELF file.

OK, we know that the size of the very entry is 0x40 bytes and I am searching for a section that named ".interp". The first 0x40 bytes of the beginning section headers table is a NULL header as you saw before. So the first 0x40 bytes is not our candidate. I select the next 0x40 bytes which is our next entry of the section headers table.

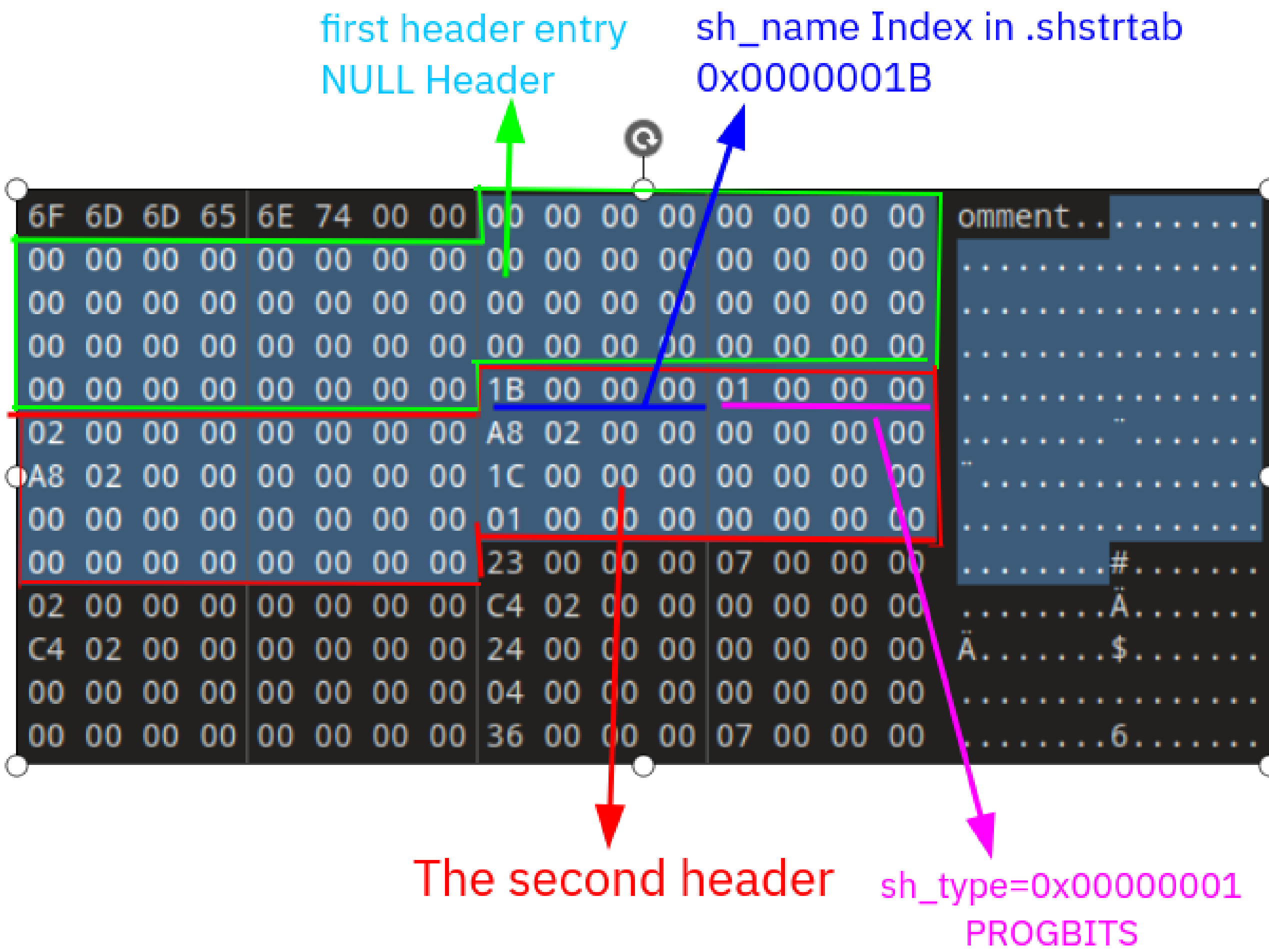

Figure 2-8: the content of the section headers table

As you see in the above image, I select 0x80 bytes from the beginning of the section headers table. The first 0x40 bytes are NULL. The second 0x40 bytes is the second entry or second section header. As said before in this part, the first 4-Byte is the index of the name of the section in the string table section. This value is 0x1B in this case.
To find the name of the section, I should first locate the string table section in this ELF file and then locate the 0x1Bth offset in it, to find the name of this section. So where is the String table (.shstrtab)? As previously mentioned, the location in question is specified in the ELF header by the Shstrndx value. The reason the string table index is in the ELF header is appears. We cannot find the string table without any information. So it should be at a particular location.

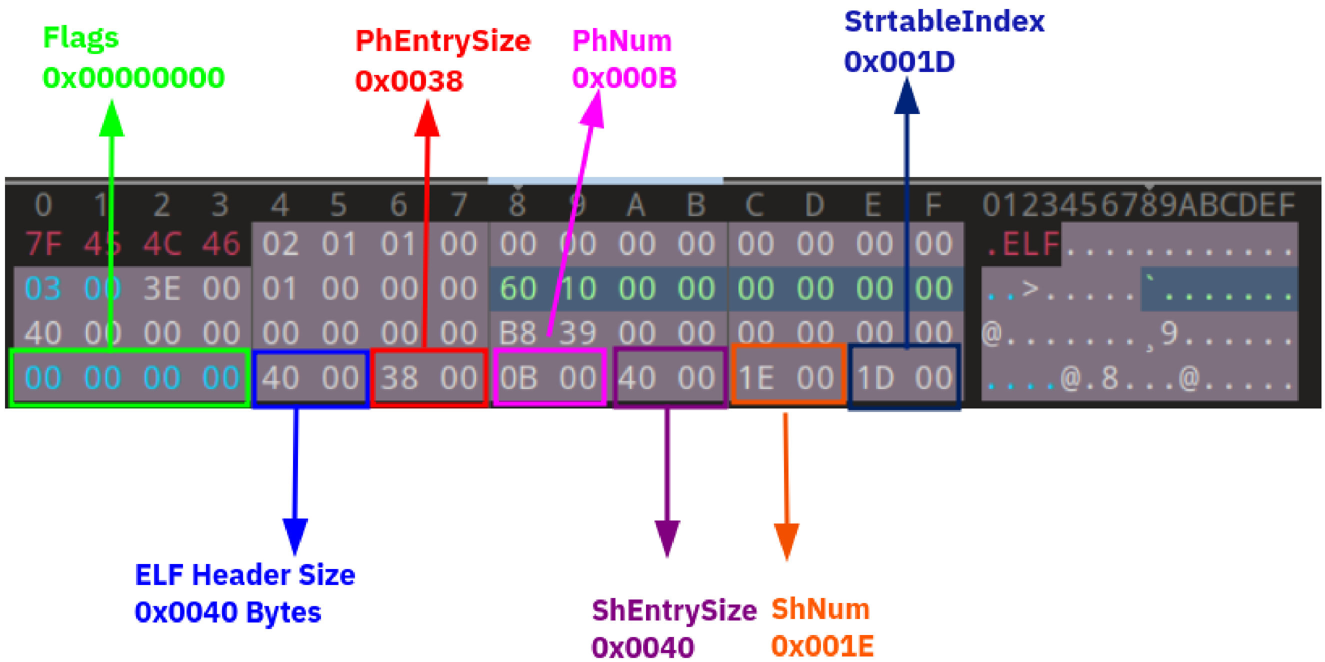

Figure 2-9: Address of Section Header String table in ELF header

As you see the index number of the Section Header String table in the section headers table is **0x1D (29)**. So we found out the entry of this section is located at the 29th table in the section headers table. To locate it we know the size of every entry that is **0x40** bytes. And now we know the index number of this entry, **0x1D**. By a simple calculation **0x40*0x1D = 0x740**

So if we go **0x740** bytes into the section headers table, we arrive at the beginning of the string table section header.

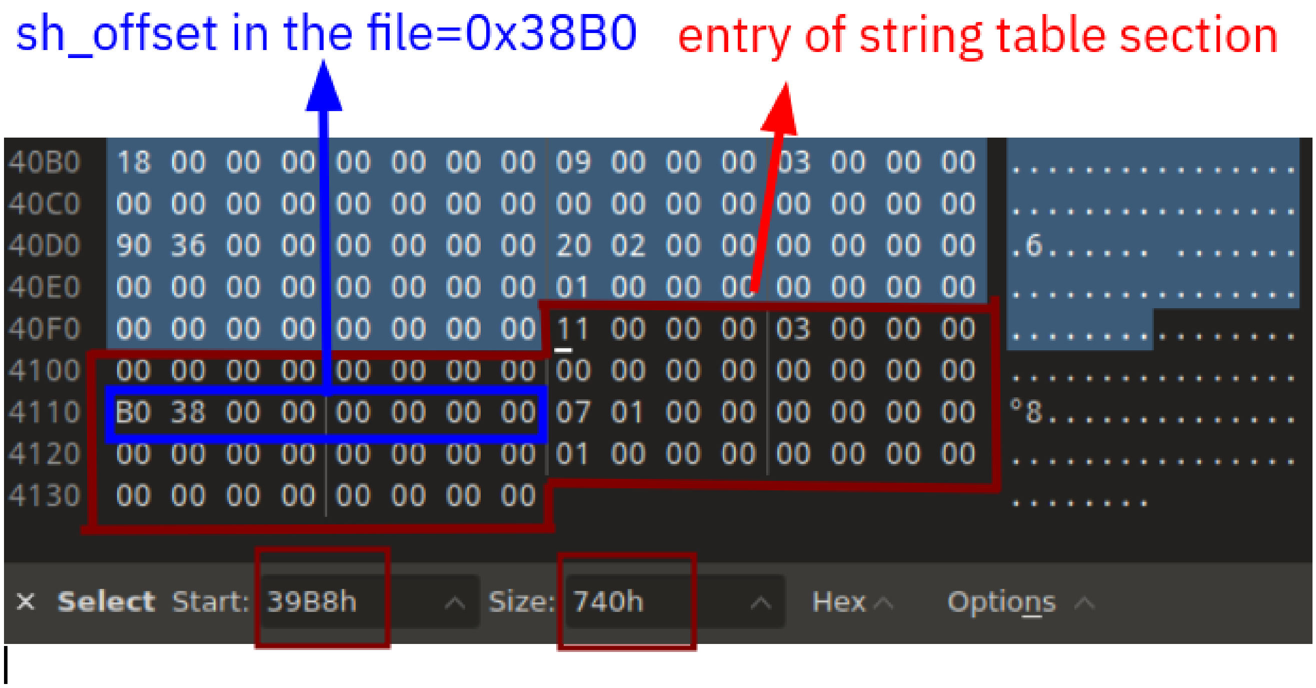

Figure 2-10: Section Header's String table

As you see the string table entry in the section headers is located at the end of the file. If you back the output of the **readelf**, see that the last row is for **shstrtab** and its index is **29**.
OK, now we have the entry of the string table but this is just its header.
Now we should find the string table location in the file. As said before the sh_offset value in the section header, is the offset of the section in the ELF file. This value for this section header is **0x38B0** as you see in the image. Now we can locate the string table at the **0x38B0 th** byte from the beginning of the ELF file.(CTRL+G)

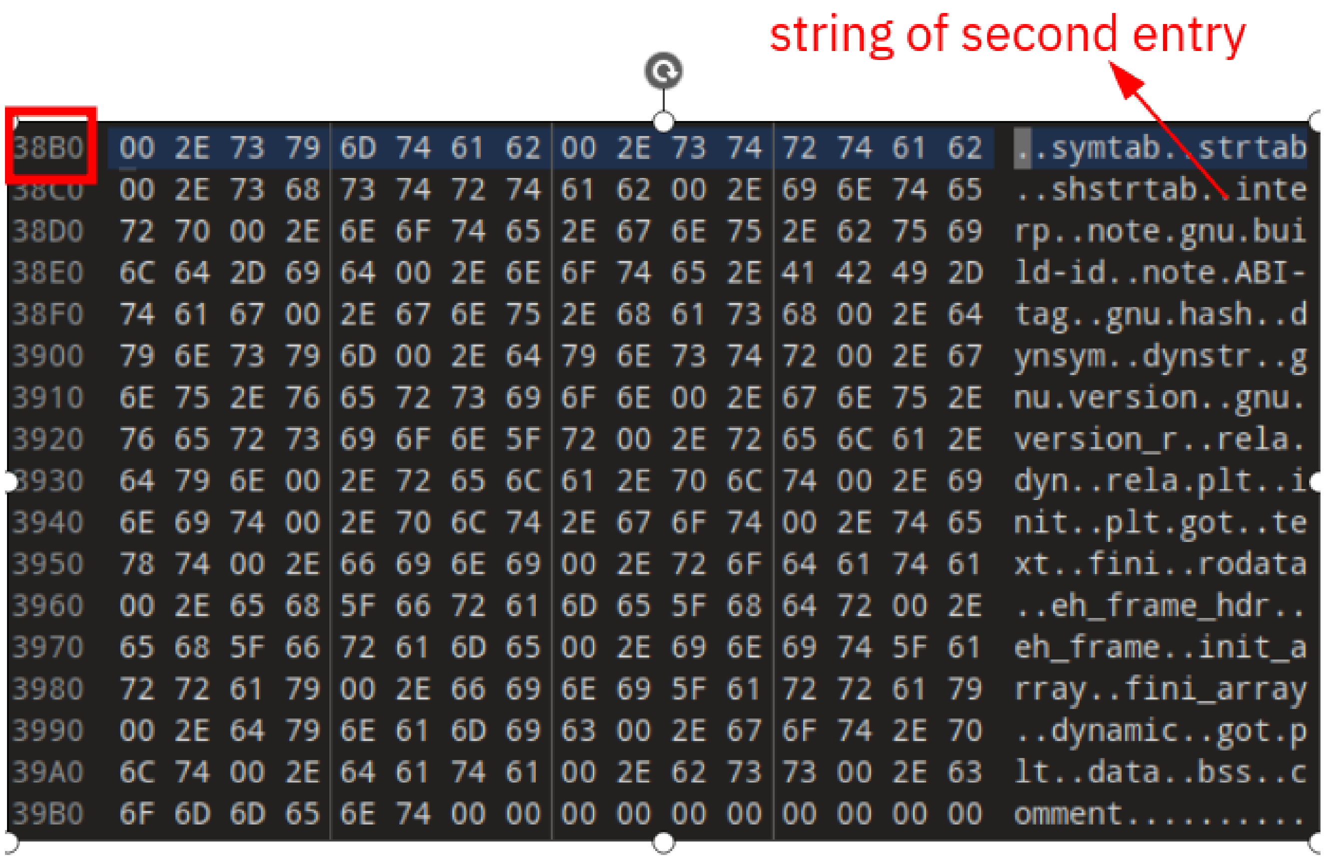

Figure 2-12: Example of shstrtab

Finally, we reached the Section Headers String table section :D.
As you see it contains some NULL-terminated strings that start with a “.”.
Now we want the name of the second section. If you remember it was at index 0x1B in the string table. So we can find it. The name is “.interp”.
Now we understand how to traverse in sections with the help of section headers and ELF header.

In this part we learned about section headers which are some tables that indicate information about sections. Also we find out how travers in the sections step by step.

If you confused, I advice to read again this part exactly and doing steps handly to understand it better.
In the next part of this story, you will learn sections in detail. Ready for it because it will be a long and hard story

Good Bye…

*