Understanding ELF using readelf and objdump

What is ELF? ELF (Executable and Linking Format) is file format that defines how an object file is composed and organized. With this information, your kernel and the binary loader know how to load the file, where to look for the code, where to look the initialized data, which shared library that needs to be loaded and so on.

First of all, you should know about different kind of ELF object:

- Relocatable file: an object file that holds code and data suitable for linking with other object files to create an executable or a shared object file. In other word, you can say that relocatable file is a foundation for creating executables and libraries.

  This is kind of file you get if you compile a source code like this:

  ```
  $ gcc -c test.c
  ```

  That will produce test.o, which is a relocatable file.

  Kernel module (either suffixed with .o or .ko) is also a form of relocatable file.

- Executable file: object file that holds a program suitable for execution. Yes, that means, your XMMS mp3 player, your vcd software player, even your text editor are all ELF executable files.

  This is also a familiar file if you compile a program:

  ```
  $ gcc -o test test.c
  ```

  After you make sure the executable bit of "test" is enabled, you can execute it. The question is, what about shell script? Shell script is NOT ELF executable, but the interpreter IS.

- Shared object file: This file holds code and data suitable for linking in two contexts:
  1. The link editor may process it with other relocatable and shared object file to create another object file.
  2. The dynamic linker combines it with an executable file and other shared objects to create a process image.

  In simple words, these are the files that you usually see with suffix .so (normally located inside /usr/lib on most Linux installation).

Is there any other way to detect the ELF type? Yes there is. In every ELF object, there is a file header that explains what kind file it is. Assuming you have installed binutils package, you can use readelf to read this header. For example (command results are shortened to show related fields only):

```
$ readelf -h /bin/ls
  Type: EXEC (Executable file)
$ readelf -h /usr/lib/crt1.o
  Type: REL (Relocatable file)
$ readelf -h /lib/libc-2.3.2.so
  Type: DYN (Shared object file)
```

"File" command works too for object file identification, but I won't discuss it further. Let's focus on readelf and objdump, since we will use both of them.

To make us easier to study ELF, you can use the following simple C program:

```
/* test.c */
#include<stdio.h>

int global_data = 4;
int global_data_2;

int main(int argc, char **argv)
{
    int local_data = 3;
    printf("Hello World\n");
    printf("global_data = %d\n", global_data);
    printf("global_data_2 = %d\n", global_data_2);
    printf("local_data = %d\n", local_data);
    return (0);
}
```

And compile it:

```
$ gcc -o test test.c
```

A. Examining ELF header.

The produced binary will be our examination target. Let’s start with the content of the ELF header:

```
$ readelf -h test
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2’s complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Intel 80386
  Version: 0x1
  Entry point address: 0x80482c0
  Start of program headers: 52 (bytes into file)
  Start of section headers: 2060 (bytes into file)
  Flags: 0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 7
  Size of section headers: 40 (bytes)
  Number of section headers: 28
  Section header string table index: 25
```

What does this header tell us?

- This executable is created for Intel x86 32 bit architecture (″machine″ and ″class″ fields).
- When executed, program will start running from virtual address 0x80482c0 (see entry point address). The ″0x″ prefix here means it is a hexadecimal number. This address doesn’t point to our main() procedure, but to a procedure named _start. Never felt you had created such thing? Of course you don’t. _start procedure is created by the linker whose purpose is to initialize your program.
- This program has a total of 28 sections and 7 segments.

What is section? Section is an area in the object file that contains information which is useful for linking: program’s code, program’s data (variables, array, string), relocation information and other. So, in each area, several information is grouped and it has a distinct meaning: code section only hold code, data section only holds initialized or non-initialized data, etc. Section Header Table (SHT) tells us exactly what sections the ELF object has, but at least by looking on "Number of section headers" field above, you can tell that "test" contains 28 sections.

If section has meaning for the binary, our Linux kernel doesn't see it the same way. The Linux kernel prepares several VMA (virtual memory area) that contains virtually contigous page frames. Inside these VMA, one or more sections are mapped. Each VMA in this case represents an ELF segment. How the kernel knows which section goes to which segment? This is the function of Program Header Table(PHT).

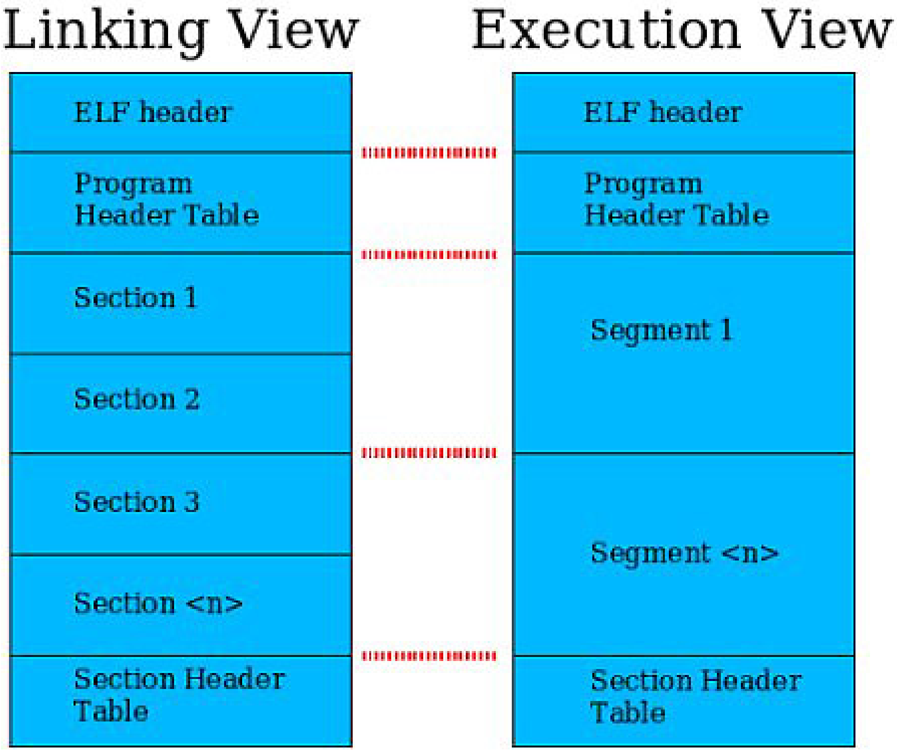


Figure 1. ELF structure in two different point of view.

B. Examining Section Header Table(SHT).

Let's see what kind of sections that exist inside our program (output is shortened):

```
$ readelf -S test
```

There are 28 section headers, starting at offset 0x80c:

```
Section Headers:
  [Nr] Name      Type     Addr     Off    Size   ES Flg Lk Inf Al
  ........
  [ 4] .dynsym   DYNSYM   08048174 000174 000060 10 A   5   1  4
  ........
  [11] .plt      PROGBITS 08048290 000290 000030 04 AX  0   0  4
  [12] .text     PROGBITS 080482c0 0002c0 0001d0 00 AX  0   0  4
  ........
  [20] .got      PROGBITS 080495d8 0005d8 000004 04 WA  0   0  4
  [21] .got.plt PROGBITS 080495dc 0005dc 000014 04 WA  0   0  4
  ........
  [22] .data     PROGBITS 080495f0 0005f0 000010 00 WA  0   0  4
  [23] .bss      NOBITS   08049600 000600 000008 00 WA  0   0  4
  ........
  [26] .symtab   SYMTAB   00000000 000c6c 000480 10     27  2c 4
  ........
```

.text section is a place where the compiler put executables code. As the consequence, this section is marked as executable ("X" on Flg field). In this section, you will see the machine codes of our main() procedure:

```
$ objdump -d -j .text test
```

-d tells objdump to diassembly the machine code and -j tells objdump to focus on specific section only (in this case, .text section)

```
  08048370:
  .......
   8048397: 83 ec 08             sub $0x8,%esp
   804839a: ff 35 fc 95 04 08    pushl 0x80495fc
   80483a0: 68 c1 84 04 08       push $0x80484c1
   80483a5: e8 06 ff ff ff       call 80482b0
   80483aa: 83 c4 10             add $0x10,%esp
   80483ad: 83 ec 08             sub $0x8,%esp
   80483b0: ff 35 04 96 04 08    pushl 0x8049604
   80483b6: 68 d3 84 04 08       push $0x80484d3
   80483bb: e8 f0 fe ff ff       call 80482b0
  .......
```

.data section hold all the initialized variable inside the program which doesn't live inside the stack. "Initialized" here means it is given an initial value like we did on "global_data". How about "local_data"? No, local_data's value isn't in .data since it lives on process's stack.

Here is what objdump found about .data section:

```
$ objdump -d -j .data test
  .....
  080495fc : 80495fc: 04 00 00 00
  .........
```

One thing that we can conclude so far is that objdump kindly does address-to-symbol transformation for us. Without looking into symbol table, we know that 0x08049424 is the address of global_data. There, we clearly see that it is initialized with 4. Please note that common executables installed by most Linux distribution has been striped out, thus there is no entry in its symbol table. It makes objdump difficult to interpret the addresses.

And what is .bss? BSS (Block Started by Symbol) is a section where all unitialized variables are mapped. You might think "everything surely has an initial value". True, in Linux case, all unitialized variables are set as zero, that's why .bss section is just bunch of zeroes. For character type variables, that means null character. Knowing this fact, we know that global_data_2 is assigned 0 on runtime:

```
$ objdump -d -j .bss test
  Disassembly of section .bss:
  .....
  08049604:
   8049604: 00 00 00 00
  .........
```

Previously, we mentioned a bit about symbol table. This table is useful to find the correlation between a symbol name (non external function, variable) and an address. Using -s, readelf will decode the symbol table for you:

```
$ readelf -s ./test
Symbol table '.dynsym' contains 6 entries:
  Num: Value    Size Type  Bind    Vis     Ndx  Name
  .....
  2:   00000000 57   FUNC GLOBAL DEFAULT   UND printf@GLIBC_2.0 (2)
  .....
Symbol table '.symtab' contains 72 entries:
  Num: Value    Size Type   Bind   Vis     Ndx  Name
  .....
  49:  080495fc 4    OBJECT GLOBAL DEFAULT 22  global_data
```

```
.....
55:  08048370 109  FUNC   GLOBAL DEFAULT 12  main
.....
59:  00000000 57   FUNC   GLOBAL DEFAULT UND printf@@GLIBC_2.0
.....
61:  08049604 4    OBJECT GLOBAL DEFAULT 23  global_data_2
.....
```

"Value" denotes the address of the symbol. For example, if an instruction refers to this address (e.g: pushl 0x80495fc), that means it refers to global_data. Printf() is treated differently, since it is a symbol that refers to an external function. Remember that printf is defined in glibc, not inside our program. Later, I will explain how our program calls printf.

C. Examining Program Header Table(PHT).

Like I explained previously, segment is the way operating system "sees" our program. Thus, let's see how will our program be segmented:

```
$ readelf -l test
.....
There are 7 program headers, starting at offset 52
Program Headers:
Type    Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align[00]
PHDR    0x000034 0x08048034 0x08048034 0x000e0 0x000e0 RE  0x4[01]
INTERP  0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1[02]
LOAD    0x000000 0x08048000 0x08048000 0x004fc 0x004fc RE  0x1000[03]
LOAD    0x0004fc 0x080494fc 0x080494fc 0x00104 0x0010c RW  0x1000[04]
DYNAMIC 0x000510 0x08049510 0x08049510 0x000c8 0x000c8 RW  0x4[05]
NOTE    0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4[06]
STACK   0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
Section to Segment mapping:
Segment Sections...
00
01      .interp
02      .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gn
u.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag
06
Note:I add numbers on the left of each PHT entries to make the reader
```

```
easier to study the section to segment mapping.
```

The mapping is quite straight forward. For example, inside segment number 02, there are 15 sections mapped. .text section is mapped in this segment. Its flags are R and E, which means it is Readable and Executable. If you see W in segment's flag, that means it is writable.

By looking on "VirtAddr" column, we can discover the virtual start address of each segment. Back to the segment number #2, the start address is 0x08048000. Later in this section, we will discover that this address isn't the real address of the segment on memory. You can ignore the PhysAddr, because in Linux always operate in protected mode (on Intel/AMD 32 bit and 64 bit) thus virtual address is the thing that matters.

Segment has many types, but let's focus on two types:

- LOAD: The segment's content is loaded from the executable file. "Offset" denotes the offset of the file where the kernel should start reading the file's content. "FileSiz" tells us how many bytes must be read from the file.

  For example, segment #2 is actually the content of the file starting from offset 0 to 4fc (offset+filesiz). To speed up the execution, the file's content is read on demand, thus it is only read from the disk if it is referenced at runtime.

- STACK : The segment is stack area. Interesting to see that all the fields except "Flg" and "Align" are given 0. Is it an error? No, it is valid. It is the kernel's job to decide where the stack segment starts from and how big it is. Remember that on Intel compatible processor, stack grows downward (address is decremented each time a value is pushed).

Courious to see the real layout of process segment? We can use /proc/<pid>/maps file to reveal it. <pid> is the PID of the process we want to observe. Before we move on, we have a small problem here. Our test program runs so fast that it ends before we can even dump the related /proc entry. I use gdb to solve this. You can use another trick such as inserting sleep() before it calls return().

In a console (or a terminal emulator such as xterm) do:

```
$ gdb test
(gdb)b main
```

```
Breakpoint 1 at 0x8048376
(gdb) r
Breakpoint 1, 0x08048376 in main ()
```

Hold right here, open another console and find out the PID of program ″test″. If you want the quick way, type:

```
$ cat /proc/`pgrep test`/maps
```

You will see an output like below (you might get different output):

```
[1] 0039d000-003b2000 r-xp 00000000 16:41 1080084 /lib/ld-2.3.3.so
[2] 003b2000-003b3000 r--p 00014000 16:41 1080084 /lib/ld-2.3.3.so
[3] 003b3000-003b4000 rw-p 00015000 16:41 1080084 /lib/ld-2.3.3.so
[4] 003b6000-004cb000 r-xp 00000000 16:41 1080085
/lib/tls/libc-2.3.3.so
[5] 004cb000-004cd000 r--p 00115000 16:41 1080085
/lib/tls/libc-2.3.3.so
[6] 004cd000-004cf000 rw-p 00117000 16:41 1080085
/lib/tls/libc-2.3.3.so
[7] 004cf000-004d1000 rw-p 004cf000 00:00 0
[8] 08048000-08049000 r-xp 00000000 16:06 66970 /tmp/test
[9] 08049000-0804a000 rw-p 00000000 16:06 66970 /tmp/test
[10] b7fec000-b7fed000 rw-p b7fec000 00:00 0
[11] bffeb000-c0000000 rw-p bffeb000 00:00 0
[12] ffffe000-fffff000 ---p 00000000 00:00 0
Note: I add number on each line as reference.
```

Back to gdb, type:

```
(gdb) q
```

So, in total, we see 12 segment (also known as Virtual Memory Area--VMA). Focus on the first and the last field. First field denotes VMA address range, while last field shows the backing file. Do you see the similarity between VMA #8 and segment #02 listed in PHT? The difference is, SHT said it is ended on 0x080484fc, but on VMA #8, we see that it ends on 0x08049000. Same thing happens between VMA #9 and segment #03; SHT said it starts at 0x080494fc, while the VMA starts at 0x0804900.

There are several facts we must observe:

1. Even though the VMA started on different address, the related sections are still mapped on exact virtual address.

2. The kernel allocate memory on per page basis and the page size is 4KB. Thus, every page address is actually a multiple of 4KB e.g: 0x1000, 0x2000 and so on. So, for the first page of VMA #9, the page's address is 0x0804900. Or technically speaking, the address of the segment is rounded down (aligned) to the nearest page boundary.

Last, which one is the stack? That is VMA #11. Usually, the kernel allocate several pages dynamically and map to the highest virtual address possible in user space to form stack area. Simply speaking, each process address space is divided into two part (this assume Intel compatible 32 bit processor): user space and kernel space. User space is in 0x00000000-0xc0000000 range, while kernel space starts on 0xc0000000 onwards.

So, it is clear that stack is assigned address range near the 0xc0000000 boundary. The end address is static, while the start address is changing according to how many values are stored on stack.

D. How a function is referenced?

If a program calls a function that resides within its own executable, all it has to do is simple: just call the procedure. But what happens if it calls something like printf() that is defined inside glibc shared library?

Here, I won't discuss deeply about how the dynamic linker really works, but I focus on how the calling mechanism is implemented inside the executable itself. With this assumption in mind, let's continue.

When a program wants to call a function, it actually does following flow:

1. It made a jump to relevant entry in PLT (Procedure Linkage Table).
2. In PLT, there is another jump to an address mentioned in related entry in GOT (Global Offset Table).
3. If this is the first the function is called, follow step #4. If this isn't, follow step #5.
4. The related GOT entry contains an address that points back to next instruction in PLT. Program will jump to this address and then calls the dynamic linker to resolve the function's address. If the function is found, its address is put in related GOT entry and then the function itself is executed.

So, another time the function is called, GOT already holds its address and PLT can jump directly to the address. This procedure is called lazy binding; all external symbols are not resolved until the time it is really needed (in this case, when a function is called). Jump to step #6.

5. Jump to the address mentioned in GOT. It is the address of the function thus PLT is no longer used.
6. Execution of the function is finished. Jump back to the next instruction in the main program.

As always, looking inside the executable is the best way to explain it. If you do:

```
$ objdump -d -j .text test
```

You will see the following code fragment:

```
.....
08048370:
.....
 804838f: e8 1c ff ff ff   call 80482b0
```

What we have on 0x80482b0 is:

```
080482b0:
 80482b0: ff 25 ec 95 04 08   jmp *0x80495ec
 80482b6: 68 08 00 00 00      push $0x8
 80482bb: e9 d0 ff ff ff      jmp 8048290   <_init+0x18>
```

As you see, the jump on 0x80482b0 is indirect jump ('*' in front of the address). So, to see where it will jump, we must peek into 0x80482b0. The guesses are, either this address is in .got section or in .got.plt. Looking back in SHT, it is clear that we must check .got.plt. I use readelf to do hexadecimal dump because it does number reordering for us:

```
$ readelf -x 21 test
Hex dump of section '.got.plt':
  0x080495dc 080482a6 00000000 00000000 08049510
  ........
  ........
  0x080495ec 080482b6
  ....
(Note: first column is virtual address. The data in this address is
```

described at the 5th column, not the second one! So, from right to left, the address is in ascending order.)

Bingo! We have "080482b6" here. In other word, we go back to PLT and there we eventually jump another address. This is where the work of the dynamic linker is started, so we will skip it. Assuming the dynamic linker has finished its magic work, the related GOT entry now holds the address of printf().

E. Alternative tool to inspect ELF structure.

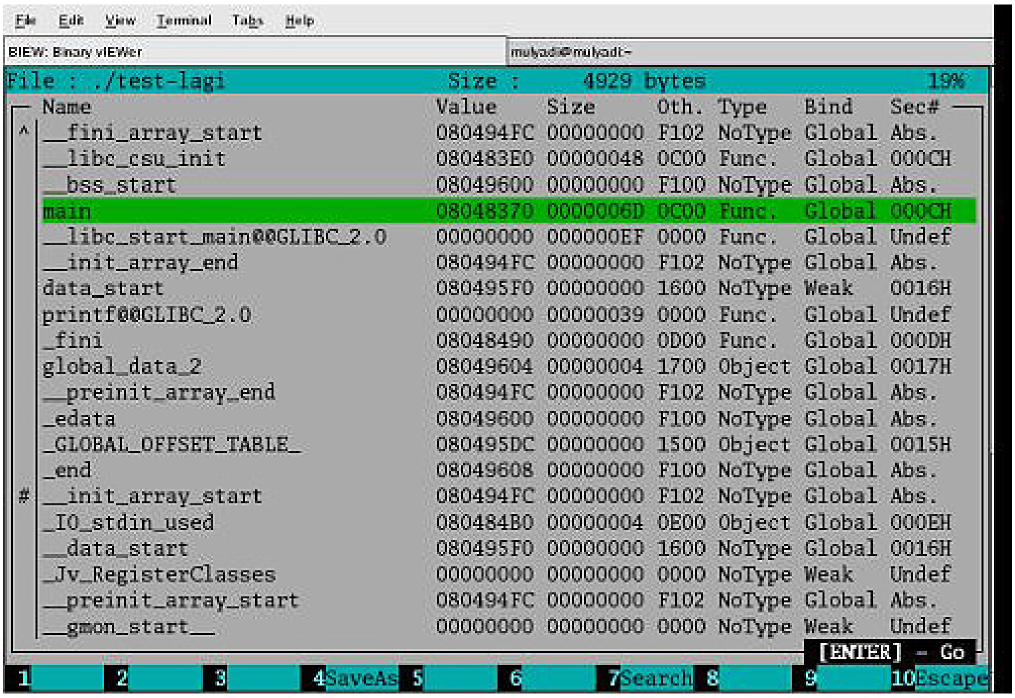
Besides counting on readelf and objdump, there is another tool called Beye. This is actually a file viewer but it is capable to parse the ELF structure.. You can grab the source from http://beye.sourceforge.net and compile it by yourself. Usually Beyeis included in hacking oriented Linux Live CD such as Phlak. Refer to the website and the packaged documents on how to compile and install Beye.

I personally like Beyebecause it offers curses based GUI display. Navigation between sections, checking ELF header, listing symbols and other tasks are now just a matter of pressing certain keyboard shortcut and you're done.

For example, you can list the symbols and directly jump to the symbol's address. Here, we try to jump to main(). First execute Beye:

```
$ beye test
```

Press Ctrl+A followed by F7 to view symbol table. To avoid wasting time traversing the table, press again F7 to open "Find string" menu. Type "main" and press Enter. Once the highlighted entry is what you're looking for, simply press Enter and Beye will jump to the address of main(). Don't forget to switch to the disassembler mode (press F2 to select it) so you can see the high level interpretation of the opcodes.


File Edit View Terminal Tabs Help
BIEW: Binary vIEWer
mulyadi@mulyadi:~
[ENTER] - Go
1 2 3 4SaveAs 5 6 7Search 8 9 10Escape

```
File : ./test-lagi                   Size :      4929 bytes              19%
 Name                              Value    Size     Oth. Type   Bind   Sec#
 __fini_array_start                080494FC 00000000 F102 NoType Global Abs.
 __libc_csu_init                   080483E0 00000048 0C00 Func.  Global 000CH
 __bss_start                       08049600 00000000 F100 NoType Global Abs.
 main                              08048370 0000006D 0C00 Func.  Global 000CH
 __libc_start_main@@GLIBC_2.0      00000000 000000EF 0000 Func.  Global Undef
 __init_array_end                  080494FC 00000000 F102 NoType Global Abs.
 data_start                        080495F0 00000000 1600 NoType Weak   0016H
 printf@@GLIBC_2.0                 00000000 00000039 0000 Func.  Global Undef
 _fini                             08048490 00000000 0D00 Func.  Global 000DH
 global_data_2                     08049604 00000004 1700 Object Global 0017H
 __preinit_array_end               080494FC 00000000 F102 NoType Global Abs.
 _edata                            08049600 00000000 F100 NoType Global Abs.
 _GLOBAL_OFFSET_TABLE_             080495DC 00000000 1500 Object Global 0015H
 _end                              08049608 00000000 F100 NoType Global Abs.
 __init_array_start                080494FC 00000000 F102 NoType Global Abs.
 _IO_stdin_used                    080484B0 00000004 0E00 Object Global 000EH
 __data_start                      080495F0 00000000 1600 NoType Global 0016H
 _Jv_RegisterClasses               00000000 00000000 0000 NoType Weak   Undef
 __preinit_array_start             080494FC 00000000 F102 NoType Global Abs.
 __gmon_start__                    00000000 00000000 0000 NoType Weak   Undef
```

Figure 2. Beye *lists all symbols*

Since we usually refer to virtual address, not file offset, it is better to switch to virtual address view. Press Ctrl+C followed by F6 and select "Local". Now, what you see in the leftmost column is the virtual address.

Conclusion

This article is just an overview on how to study ELF structure. Using readelf and objdump, you are ready to take your first journey. If needed, tool like Beye can help you to explore the binary internal faster. Use any arsenal you have, be creative and practice it regularly, then soon you can master the technique. Happy exploring.

Further reading

- http://www.linuxjournal.com/article/1059
  http://www.linuxjournal.com/article/1060

  Two good ELF introductory articles written by Eric Youngdale.

- http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

  Explanation about ELF from Wikipedia. From there, you can find links to another useful documents.

- http://x86.ddj.com/ftp/manuals/tools/elf.pdf

  The document that completely explain all about ELF structure. Study this document after reading this article to gain complete insight about ELF.

- ELFSH

  A tool to do ELF binary inspection and manipulation. Pretty useful for reverse engineering too. It has scripting feature so you can automate most of your work. In the website, there are many documents that explains various ELF hacking.