Linux ELF Header Basics for Incident Responders

Taz Wake | Halkyn Consulting Ltd | Originally published 2024-12-30

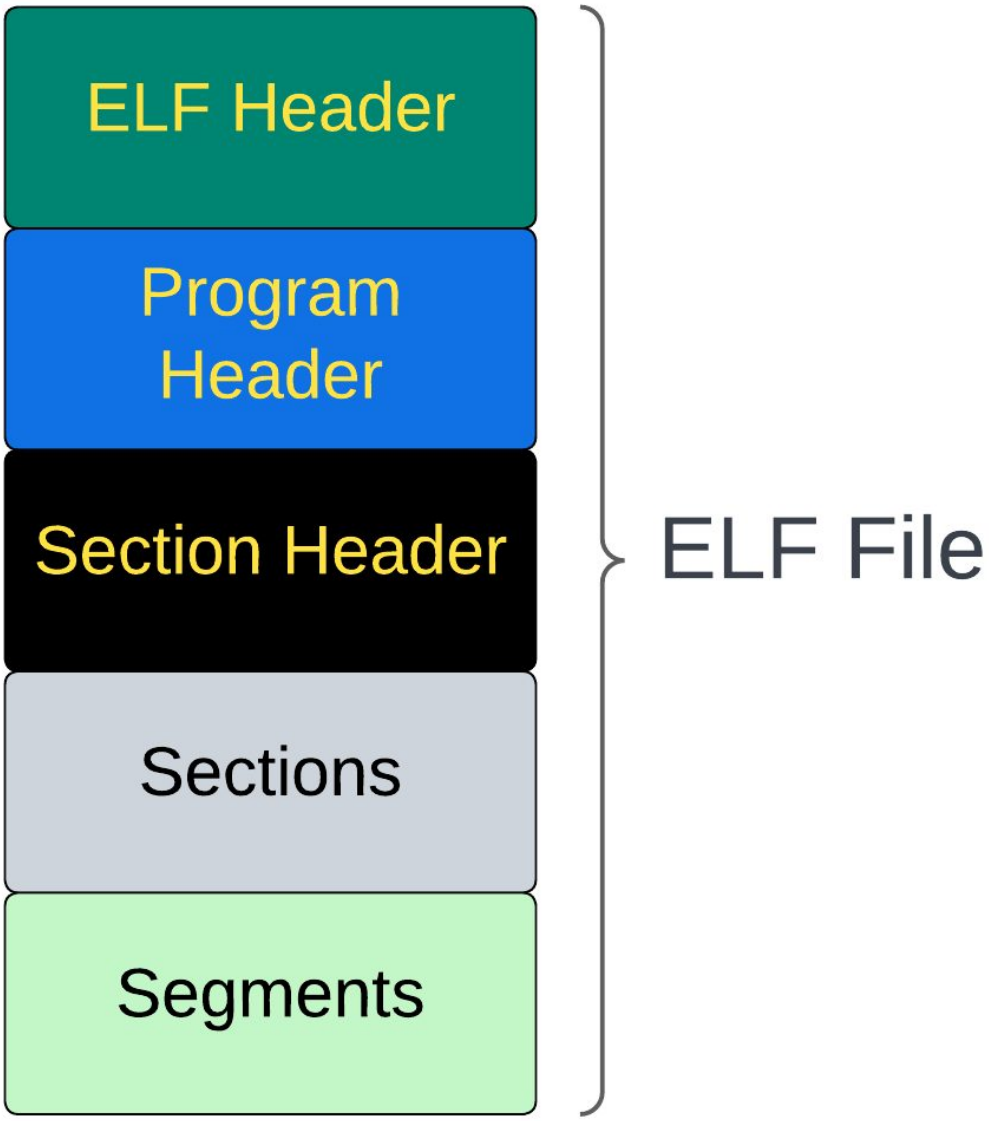
Despite some strange ideas on social media platforms, and despite the fact that Linux-based operating systems really can be great, there is no escaping the fact that we still have to deal with malware and intrusions on every platform. There is no such thing as a "secure OS." At some point, Incident Response is going to be needed.

One thing that can slow down IR, is dealing with unfamiliar file structures. Most IR professionals understand MS Windows PE headers and tend to be pretty good at getting information out of them. With the ELF format used by Linux executables, it isn't so simple.

This article is a quick look at the basic structures that make up an ELF file header, so you can understand the structure a bit more. It isn't a replacement for learning to reverse engineer malware!

ELF Structure

An ELF file contains the following components:



ELF File Structure

ELF Header:

- Located at the start of the file.
- Contains metadata about the file, including architecture, entry point, and offsets to other parts of the file.

Program Header Table (Optional for object files):

- Describes segments of the file to be loaded into memory.
- Used by the loader to map the file into a process's memory space.

Section Header Table (Optional for executables):

- Describes sections in the file, such as .text, .data, .bss, and .symtab.
- Used by linkers and debugging tools but typically not required for execution.

Sections:

- Contain specific types of data, like executable code, initialized data, uninitialized data, or debugging information.

Segments:

- Represent contiguous regions of memory and are described in the Program Header Table.
- Includes regions like text (code), data (initialized variables), and bss (uninitialized variables).

This article will focus on the ELF Header, which is the first structure.

Generally, the execution flow, for a Linux executable, goes something like:

- **ELF Header**: Provides metadata to locate the Program Header Table and entry point.
- **Program Header Table**: Guides the loader on how to map the file into memory.
- **Sections**: Contain the actual code, data, and linking/debugging information.

ELF Header

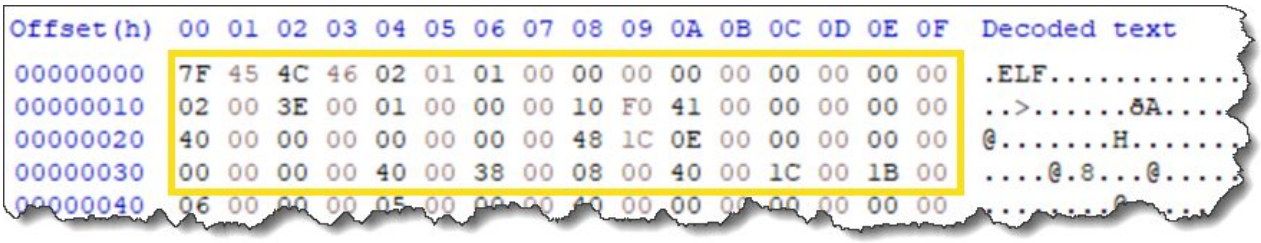
This table shows the structure of the ELF header. This is 64 bytes for a 64-bit file and 52 bytes for a 32-bit ELF file. The extra size accounts for the increased size of some fields which deal with the increased 64-bit address space.

Offset	Size (Bytes)	Field Name	Description
0x00	16	e_ident	Identification bytes, including magic number (\x7fELF), class, and endianness.
0x10	2	e_type	File type: relocatable, executable, shared, or core.
0x12	2	e_machine	Architecture type (e.g., x86, x86-64, ARM, etc.).
0x14	4	e_version	ELF version, usually 1 (original version).
0x18	4 (32-bit) / 8 (64-bit)	e_entry	Entry point virtual address of the executable code.
0x1C (32-bit) / 0x20 (64-bit)	4 (32-bit) / 8 (64-bit)	e_phoff	Offset of the program header table in bytes.
0x20 (32-bit) / 0x28 (64-bit)	4 (32-bit) / 8 (64-bit)	e_shoff	Offset of the section header table in bytes.
0x24 (32-bit) / 0x30 (64-bit)	4	e_flags	Flags specific to the target architecture.
0x28 (32-bit) / 0x34 (64-bit)	2	e_ehsize	Size of this ELF header in bytes.
0x2A (32-bit) / 0x36 (64-bit)	2	e_phentsize	Size of one entry in the program header table.
0x2C (32-bit) / 0x38 (64-bit)	2	e_phnum	Number of entries in the program header table.
0x2E (32-bit) / 0x3A (64-bit)	2	e_shentsize	Size of one entry in the section header table.
0x30 (32-bit) / 0x3C (64-bit)	2	e_shnum	Number of entries in the section header table.
0x32 (32-bit) / 0x3E (64-bit)	2	e_shstrndx	Index of the section header string table.

ELF Header Fields and Offsets

Worked Example

This is an ELF header for a 64-bit Linux executable (actually bash) viewed in a hex editor.



ELF file - SHA256 hash: 0146420bfadda5d983ef52e52ab07adc53a0c2abe52f6e2b2648607da02dd65e

Now it is true that, most of the time, you will use tools to parse the structures, it is useful to understand them and to be able to recognise elements. In DFIR engagements you

might encounter fragments of an ELF file (especially if a threat actor has tried to delete it), or be faced with environments where your usual tooling is unavailable.

So, let's dig in.

Line 1 - Bytes 0 - 15

- **e_ident**. This is the 16-byte field starting at offset 0. This is broken down into sub-fields.

Line 1 - bytes 0 - 15

The sub-fields are:

1. **Magic Bytes - 4 bytes**. The string 0x7f454c46, or .ELF in ASCII, confirm this is a Linux ELF format file.
2. **Architecture - 1 byte**. This specifies the type of file. A 0x01 is a 32-bit file. This file uses 0x02, indicating it is a 64-bit file.
3. **Endianness - 1 byte**. Here the 0x01 indicates this is a little-endian file. A 0x02 indicates big-endian.
4. **ELF Version - 1 byte**. This is a 01 indicating ELF version 1, which is the current standard.
5. **OS Target - 1 byte**. The 8th byte specifies the target operating system (or ABI - Application Binary Interface). This file has 0x00, which is the default and indicates no specific ABI extensions are required. Other values include 0x02 for NetBSD, 0x07 for AIX and 0x09 for FreeBSD.
6. **ABI Version - 1 byte**. The 9th byte relates to any specific factors relating to the ABI selected in the 8th byte. Here it is 0x00, but this is normal because no ABI was selected.
7. **Padding - 7 bytes**. This is currently unused space and should be set to 0x00. It is available for use in future versions of the ELF standard.

Line 2 - Bytes 16 - 31

Line 2 - bytes 16 - 32

In a 64-bit file, there are four objects on this line.

- **e_type - 2 bytes**. This indicates the object file type. A 0x0 is no file type, 0x1 is relocatable, 0x2 is executable, 0x3 is a shared object and 0x4 is a core file. Note,

remember to check the endianess. In the example here, this field reads 0x0200, which is little endian for 0x2.

- **e_machine - 2 bytes**. The data here shows the required architecture. This allows ELF files to work on a range of Unix/Linux platforms, which, in turn, means there are a lot of options. The most common value on Linux today is probably 0x3E (62 decimal) which means AMD x86-64 architectures. Other examples include 0x3 for Intel 80386; 0x15 (21 decimal) for 64-bit Power PC; and 0x32 (50 decimal) for Intel IA-64 processor architectures. The bash example we have here has 0x3e00 in this field, which indicates AMD x86-64 architecture.
- **e_version - 4 bytes**. The only, currently, valid string here is 0x1. This is the ELF file format version.
- **e_entry - 8 bytes**. This field stores the virtual memory address for the program's entry point. This is where the program would begin execution, and the example here is 0x41F010.

Line 3 - Bytes 32 - 47

```
4000 0000 0000 0000 481c 0e00 0000 0000
```

Line 3 - bytes 32 - 47

There are two objects here in a 64-bit ELF file.

- **e_phoff - 8 bytes**. This identifies where the program header table starts inside the file. In the example here (little-endian) it would be 0x40, or 64 bytes into the file, and immediately after the ELF header.
- **e_shoff - 8 byes**. This identifies where the section header table starts, showing the locations of sections such as .text, .data, .bss etc. In the example here, it is 0xE1C48, or decimal. If you go to this location in the file, you will find the start of the section header table, which is always a 64-byte NULL field, followed by the individual sections.

The first two entries in the section header table for the bash binary under investigations

---

Line 4 - Bytes 48-63

Line 3 - bytes 48 - 63

This is 7 fields in a 64-bit ELF.

- **e_flags - 4 bytes**. This is unused in x86-64 and would normally be 0x00s.
- **e_ehsize - 2 bytes**. This shows the ELF header size. The example here is 0x0040, which indicates a 64-byte header.
- **e_phentsize - 2 bytes**. Size of the program header entry. In the example we are using it is 0x0038, or 56 bytes.
- **e_phnum** - 2 bytes. This shows the number of program header entries. Here we have 0x0008, or 8 entries.
- **e_shentsize** - 2 bytes. This reports the size of each section header entry. Our example is 0x0040, or 64-byte section headers.
- **e_shnum - 2 bytes**. This field reports the number of section headers. 0x001C indicates that there are 28 entries in the bash file.
- **e_shstrndx - 2 bytes**. The index of the section header string table. In our example, this is 0x001B or 27.

Using tools

Although it is important that digital forensic investigators understand how to get this data from the raw files (otherwise how do we know when our tools are mistaken), this information is available through a variety of tooling.

The most basic is the **file** command.

```
elf-Linux-x64-bash: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynam
ically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.18,
BuildID[sha1]=3fbc74b2edb4842f494d8dd682ea61c60816eaf1, stripped
```

Running file on the binary

We can also use **exiftool**, although this is not installed by default on most Linux distros.

```
ExifTool Version Number         : 12.40
File Name                       : elf-Linux-x64-bash
Directory                       : .
File Size                       : 905 KiB
File Modification Date/Time     : 2024:12:27 17:12:05+00:00
File Access Date/Time           : 2024:12:27 17:12:25+00:00
File Inode Change Date/Time     : 2024:12:27 17:12:05+00:00
File Permissions                : -rwxr-xr-x
File Type                       : ELF executable
File Type Extension             :
MIME Type                       : application/octet-stream
CPU Architecture                : 64 bit
CPU Byte Order                  : Little endian
Object File Type                : Executable file
CPU Type                        : AMD x86-64
```

Running exiftool on the binary

Another good tool for analysing Linux executables is the **readelf** command.

Using the -h switch gives a full, interpreted, output from the header.

```
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:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x41f010
  Start of program headers:          64 (bytes into file)
  Start of section headers:          924744 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

The output of readelf -h

You can directly review the section headers with the -S argument. This should match the data we identified in the raw hex.

```
There are 28 section headers, starting at offset 0xe1c48:

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        0000000000400200 000200 00001c 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            000000000040021c 00021c 000020 00   A  0   0  4
  [ 3] .note.gnu.build-id NOTE            000000000040023c 00023c 000024 00   A  0   0  4
  [ 4] .hash             HASH            0000000000400260 000260 004134 04   A  6   0  8
  [ 5] .gnu.hash         GNU_HASH        0000000000404398 004398 003648 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000004079e0 0079e0 00c690 18   A  7   1  8
  [ 7] .dynstr           STRTAB          0000000000414070 014070 008008 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          000000000041c078 01c078 00108c 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         000000000041d108 01d108 000070 00   A  7   2  8
  [10] .rela.dyn         RELA            000000000041d178 01d178 0000c0 18   A  6   0  8
  [11] .rela.plt         RELA            000000000041d238 01d238 0011d0 18   A  6  13  8
  [12] .init             PROGBITS        000000000041e408 01e408 000018 00  AX  0   0  4
  [13] .plt              PROGBITS        000000000041e420 01e420 000bf0 10  AX  0   0  4
  [14] .text             PROGBITS        000000000041f010 01f010 08a218 00  AX  0   0 16
  [15] .fini             PROGBITS        00000000004a9228 0a9228 00000e 00  AX  0   0  4
  [16] .rodata           PROGBITS        00000000004a9240 0a9240 01d310 00   A  0   0 32
  [17] .eh_frame_hdr     PROGBITS        00000000004c6550 0c6550 003b54 00   A  0   0  4
  [18] .eh_frame         PROGBITS        00000000004ca0a8 0ca0a8 00eb74 00   A  0   0  8
  [19] .ctors            PROGBITS        00000000006d9000 0d9000 000010 00  WA  0   0  8
  [20] .dtors            PROGBITS        00000000006d9010 0d9010 000010 00  WA  0   0  8
  [21] .jcr              PROGBITS        00000000006d9020 0d9020 000008 00  WA  0   0  8
  [22] .dynamic          DYNAMIC         00000000006d9028 0d9028 0001c0 10  WA  7   0  8
  [23] .got              PROGBITS        00000000006d91e8 0d91e8 000008 08  WA  0   0  8
  [24] .got.plt          PROGBITS        00000000006d91f0 0d91f0 000608 08  WA  0   0  8
  [25] .data             PROGBITS        00000000006d9800 0d9800 008360 00  WA  0   0 32
  [26] .bss              NOBITS          00000000006e1b60 0e1b60 005a48 00  WA  0   0 32
  [27] .shstrtab         STRTAB          0000000000000000 0e1b60 0000e5 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)
```

The output of readelf -S

Summary

It is important for DIFR investigators/responders to understand the structure of an ELF file. A good place to start is the header as this contains details about how the rest of the file would be structured and tells us where to look to find things like the .bss, .data or .text sections.

Most of the time we can use our tools to recover this information, but it definitely helps if you understand how to validate your tools and how to recover data from damaged files.

About Halkyn Consulting

Halkyn Consulting Ltd is a UK-based security and risk management consultancy providing 24/7 incident response, digital forensics, threat hunting, virtual security leadership (vCISO) and security compliance support to organisations worldwide. We deliver clear, practical, vendor-independent advice based on real-world threats — no hype, no upsell.

More free security resources are available at www.halkynconsulting.co.uk/security-resources.

Web: www.halkynconsulting.co.uk | **Email:** info@halkynconsulting.co.uk | **Tel:** +44 1244 940858