


The 101 of ELF files on Linux: Understanding and Analysis

Michael Boelen

- [What is an ELF file?](#)
 - [Why learn the details of ELF?](#)
 - [From source to process](#)
 - [Before you start](#)
- [The anatomy of an ELF file](#)
 - [Structure](#)
 - [ELF header](#)
 - [File data](#)
 - [Program headers](#)
 - [ELF sections](#)
 - [Static versus Dynamic binaries](#)
- [Tools for binary analysis](#)
 - [Popular tools](#)
 - [Radare2](#)
 - [Software packages](#)
 - [Example binary file](#)
- [Frequently Asked Questions](#)
 - [What is ABI?](#)
 - [What is ELF?](#)
 - [How can I see the file type of an unknown file?](#)
- [Conclusion](#)
- [More resources](#)

This article has last been updated at January 28, 2025.

Some of the true craftsmanship in the world we take for granted. One of these things is the common tools on Linux, like *ps* and *ls*. Even though the commands might be perceived as simple, there is more to it when looking under the hood. This is where [ELF](#)  or the *Executable and Linkable Format* comes in. A file format that used a lot, yet truly understood by only a few. Let's get this understanding with this introduction tutorial!

By reading this guide, you will learn:

- Why ELF is used and for what kind of files
- Understand the structure of ELF and the details of the format
- How to read and analyze an ELF file such as a binary
- Which tools can be used for binary analysis

What is an ELF file?

ELF is the abbreviation for **Executable and Linkable Format** and defines the structure for binaries, libraries, and core files. The formal specification allows the operating system to interpret its underlying machine instructions correctly. ELF files are typically the output of a compiler or linker and are a binary format. With the right tools, such file can be analyzed and better understood.

Why learn the details of ELF?

Before diving into the more technical details, it might be good to explain why an understanding of the ELF format is useful. As a starter, it helps to learn the inner workings of our operating system. When something goes wrong, we might better understand what happened (or why). Then there is the value of being able to research ELF files, especially after a security breach or discover suspicious files. Last but not least, for a better understanding while developing. Even if you program in a high-level language like Golang, you still might benefit from knowing what happens behind the scenes.

So why learn more about ELF?

- Generic understanding of how an operating system works
- Development of software
- Digital Forensics and Incident Response (DFIR)
- Malware research (binary analysis)

From source to process

So whatever operating system we run, it needs to translate common functions to the language of the CPU, also known as machine code. A function could be something basic like opening a file on disk or showing something on the screen. Instead of talking directly to the CPU, we use a programming language, using internal functions. A compiler then translates these functions into object code. This object code is then linked into a full program, by using a linker tool. The result is a binary file, which then can be executed on that specific platform and CPU type.

Before you start

This blog post will share a lot of commands. Don't run them on production systems. Better do it on a test machine. If you like to test commands, copy an existing binary and use that. Additionally, we have provided a small C program, which can you compile. After all, trying out is the best way to learn and compare results.

The anatomy of an ELF file

A common misconception is that ELF files are just for binaries or executables. We already have seen they can be used for partial pieces (object code). Another example is *shared libraries* or even *core dumps* (those core or a.out files). The ELF specification is also used on Linux for the kernel itself and Linux kernel modules.

```
michael@notebook:~$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
for GNU/Linux 2.6.24, BuildID[sha1]=2053194ca4ee8754c695f5a7a7cff2fb8fdd
```

The file command shows some basics about this binary file

Structure

Due to the extensible design of ELF files, the structure differs per file. An ELF file consists of:

1. ELF header
2. File data

With the *readelf* command, we can look at the structure of a file and it will look something like this:

```

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:                 0x4013e2
  Start of program headers:            64 (bytes into file)
  Start of section headers:           25376 (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:          28
  Section header string table index: 27

```

linux-audit.com

Details of an ELF binary


As can be seen in this screenshot, the ELF header starts with some magic. This ELF header magic provides information about the file. The first four hexadecimal parts define that this is an ELF file (45=E,4c=L,46=F), prefixed with the 7f value.

This ELF header is mandatory. It ensures that data is correctly interpreted during linking or execution. To better understand the inner working of an ELF file, it is useful to know this header information is used.

Class

After the ELF type declaration, there is a Class field defined. This value determines the architecture for the file. It can be a **32-bit** (=01) or **64-bit** (=02) architecture. The magic shows a 02, which is translated by the readelf command as an ELF64 file. In other words, an ELF file using the 64-bit architecture. Not surprising, as this particular machine contains a modern CPU.

Data

Next part is the data field. It knows two options: 01 for **LSB** [Least Significant Bit](#) , also known as little-endian. Then there is the value 02, for **MSB** (Most Significant Bit, big-endian). This particular value helps to interpret the remaining objects correctly within the file. This is important, as different types of processors deal differently with the incoming instructions and data structures. In this case, LSB is used, which is common for AMD64 type processors.

The effect of LSB becomes visible when using hexdump on a binary file. Let's show the ELF header details for /bin/ps.

```

$ hexdump -n 16 /bin/ps
00000000 457f 464c 0102 0001 0000 0000 0000 0000

00000010

```

We can see that the value pairs are different, which is caused by the right interpretation of the byte order.

Version

Next in line is another “01” in the magic, which is the version number. Currently, there is only 1 version type: currently, which is the value “01”. So nothing interesting to remember.

OS/ABI

Each operating system has a big overlap in common functions. In addition, each of them has specific ones, or at least minor differences between them. The definition of the right set is done with an **Application Binary Interface** ([ABI](#)). This way the operating system and applications both know what to expect and functions are correctly forwarded. These two fields describe what ABI is used and the related version. In this case, the value is 00, which means no specific extension is used. The output shows this as [System V](#).

ABI version

When needed, a version for the ABI can be specified.

Machine

We can also find the expected machine type (AMD64) in the header.

Type

The **type** field tells us what the purpose of the file is. There are a few common file types.

- CORE (value 4)
- DYN (Shared object file), for libraries (value 3)
- EXEC (Executable file), for binaries (value 2)
- REL (Relocatable file), before linked into an executable file (value 1)

While some of the fields could already be displayed via the magic value of the *readelf* output, there is more. For example for what specific processor type the file is. Using *hexdump* we can see the full ELF header and its values.

```
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 | .ELF.....|
02 00 3e 00 01 00 00 00 a8 2b 40 00 00 00 00 00 | ..<.....+@....|
40 00 00 00 00 00 00 00 30 65 01 00 00 00 00 00 | @.....0e.....|
00 00 00 00 40 00 38 00 09 00 40 00 1c 00 1b 00 | ....@.8...@....|
```

(output created with *hexdump -C -n 64 /bin/ps*)

The highlighted field above is what defines the machine type. The value 3e is 62 in decimal, which equals to AMD64.

While you can do a lot with a hexadecimal dump, it makes sense to let tools do the work for you. The **dumpelf** tool can be helpful in this regard. It shows a formatted output very similar to the ELF header file. Great to learn what fields are used and their typical values.

With all these fields clarified, it is time to look at where the real magic happens and move into the next headers!

File data

Besides the ELF header, ELF files consist of three parts.

- **Program Headers or Segments** (9)

- **Section Headers or Sections (28)**
- **Data**

Before we dive into these headers, it is good to know that ELF has two complementary “views”. One is to be used for the linker to allow execution (segments). The other one for categorizing instructions and data (sections). So depending on the goal, the related header types are used. Let’s start with program headers, which we find on ELF binaries.

An ELF file consists of zero or more segments, and describe how to create a process/memory image for runtime execution. When the kernel sees these segments, it uses them to map them into virtual address space, using the `mmap(2)` system call. In other words, it converts predefined instructions into a memory image. If your ELF file is a normal binary, it requires these program headers. Otherwise, it simply won’t run. It uses these headers, with the underlying data structure, to form a process. This process is similar for shared libraries.

```
Elf file type is EXEC (Executable file)
Entry point 0x402ba8
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz              MemSiz              Flags   Align

  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E     8

  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R       1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]

  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000001514 0x0000000000001514  R E    200000

  LOAD           0x00000000000015e0 0x0000000000615e00 0x0000000000615e00
                 0x00000000000005f8 0x000000000000214e8  RW     200000

  DYNAMIC        0x00000000000015e18 0x0000000000615e18 0x0000000000615e18
                 0x00000000000001e0 0x00000000000001e0  RW      8

  NOTE          0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R       4

  GNU_EH_FRAME   0x00000000000012c84 0x0000000000412c84 0x0000000000412c84
                 0x0000000000000071c 0x0000000000000071c  R       4

  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

  GNU_RELRO      0x00000000000015e00 0x0000000000615e00 0x0000000000615e00
                 0x0000000000000200 0x0000000000000200  R       1

Section to Segment mapping:
Segment Sections...
00
01      .interp
02      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dyn
.gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_fr
03      .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag .note.gnu.build-id
06      .eh_frame_hdr
07
08      .init_array .fini_array .jcr .dynamic .got
```

An overview of program headers in an ELF binary

We see in this example that there are 9 program headers. When looking at it for the first time, it's hard to understand what happens here. So let's go into a few details.

GNU_EH_FRAME

This is a sorted queue used by the GNU C compiler (gcc). It stores exception handlers. So when something goes wrong, it can use this area to deal correctly with it.

GNU_STACK

This header is used to store stack information. The stack is a buffer, or scratch place, where items are stored, like local variables. This will occur with LIFO (Last In, First Out), similar to putting boxes on top of each other. When a process function is started a block is reserved. When the function is finished, it will be marked as free again. Now the interesting part is that a stack shouldn't be executable, as this might introduce security vulnerabilities. By manipulation of memory, one could refer to this executable stack and run intended instructions.

If the GNU_STACK segment is not available, then usually an executable stack is used. The `scanelf` and `execstack` tools are two examples to show the stack details.

```
$ scanelf -e /bin/ps
TYPE   STK/REL/PTL FILE
ET_EXEC RW- R-- RW- /bin/ps

$ execstack -q /bin/ps
- /bin/ps
```

Commands to see program headers

- `dumpelf` (pax-utils)
- `elfls -S /bin/ps`
- `eu-readelf -program-headers /bin/ps`

ELF sections

The section headers define all the sections in the file. As said, this “view” is used for linking and relocation.

Sections can be found in an ELF binary after the GNU C compiler transformed C code into assembly, followed by the GNU assembler, which creates objects of it.

As the image above shows, a segment can have 0 or more sections. For executable files there are four main sections: **.text**, **.data**, **.rodata**, and **.bss**. Each of these sections is loaded with different access rights, which can be seen with **readelf -S**.

.text

Contains executable code. It will be packed into a segment with read and execute access rights. It is only loaded once, as the contents will not change. This can be seen with the `objdump` utility.

```
12 .text 0000a3e9 0000000000402120 0000000000402120 00002120 2**4
CONTENTS, ALLOC, LOAD, READONLY, CODE
```

.data

Initialized data, with read/write access rights

.rodata

Initialized data, with read access rights only (=A).

.bss

Uninitialized data, with read/write access rights (=WA)

```
[24] .data PROGBITS 000000000006172e0 000172e0
00000000000000100 0000000000000000 **WA** 0 0 8
[25] .bss NOBITS 000000000006173e0 000173e0
0000000000021110 0000000000000000 **WA** 0 0 32
```

Commands to see section and headers

- dumpelf
- elfls -p /bin/ps
- eu-readelf -section-headers /bin/ps
- readelf -S /bin/ps
- objdump -h /bin/ps

Section groups

Some sections can be grouped, as they form a whole, or in other words be a dependency. Newer linkers support this functionality. Still, this is not common to find that often:

```
$ readelf -g /bin/ps
There are no section groups in this file.
```

While this might not be looking very interesting, it shows a clear benefit of researching the ELF toolkits which are available, for analysis. For this reason, an overview of tools and their primary goal have been included at the end of this article.

Static versus Dynamic binaries

When dealing with ELF binaries, it is good to know that there are two types and how they are linked. The type is either static or dynamic and refers to the libraries that are used. For optimization purposes, we often see that binaries are “dynamic”, which means it needs external components to run correctly. Often these external components are normal libraries, which contain common functions, like opening files or creating a network socket. Static binaries, on the other hand, have all libraries included. It makes them bigger, yet more portable (e.g. using them on another system).

If you want to check if a file is statically or dynamically compiled, use the `file` command. If it shows something like:

```
$ file /bin/ps
/bin/ps: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), **dynamically linked (uses shared
```

To determine what external libraries are being used, simply use the `ldd` command on the same binary:


```
$ ldd /bin/ps
linux-vdso.so.1 => (0x00007ffe5ef0d000)
libprocps.so.3 => /lib/x86_64-linux-gnu/libprocps.so.3 (0x00007f8959711000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f895934c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8959935000)
```

Tip: To see underlying dependencies, it might be better to use the **lddtree** utility instead.

When you want to analyze ELF files, it is definitely useful to look first for the available tooling. Some of the software packages available provide a toolkit to reverse engineer binaries or executable code. If you are new to analyzing ELF malware or firmware, consider learning **static analysis** first. This means that you inspect files without actually executing them. When you better understand how they work, then move to **dynamic analysis**. Now you will run the file samples and see their actual behavior when the low-level code is executed as actual processor instructions. Whatever type of analysis you do, make sure to do this on a dedicated system, preferably with strict rules regarding networking. This is especially true when dealing with unknown samples or those are related to malware.

Popular tools

Radare2

The [Radare2](#)  toolkit has been created by Sergi Alvarez. The '2' in the version refers to a full rewrite of the tool compared with the first version. It is nowadays used by many reverse engineers to learn how binaries work. It can be used to dissect firmware, malware, and anything else that looks to be in an executable format.

Software packages


Most Linux systems will already have the binutils package installed. Other packages might help with showing much more details. Having the right toolkit might simplify your work, especially when doing analysis or learning more about ELF files. So we have collected a list of packages and the related utilities in it.

elfutils

- /usr/bin/eu-addr2line
- /usr/bin/eu-ar - alternative to ar, to create, manipulate archive files
- /usr/bin/eu-elfcmp
- /usr/bin/eu-elflint - compliance check against gABI and psABI specifications
- /usr/bin/eu-findtextrel - find text relocations
- /usr/bin/eu-ld - combining object and archive files
- /usr/bin/eu-make-debug-archive
- /usr/bin/eu-nm - display symbols from object/executable files
- /usr/bin/eu-objdump - show information of object files
- /usr/bin/eu-ranlib - create index for archives for performance
- /usr/bin/eu-readelf - human-readable display of ELF files
- /usr/bin/eu-size - display size of each section (text, data, bss, etc)
- /usr/bin/eu-stack - show the stack of a running process, or coredump
- /usr/bin/eu-strings - display textual strings (similar to strings utility)
- /usr/bin/eu-strip - strip ELF file from symbol tables
- /usr/bin/eu-unstrip - add symbols and debug information to stripped binary

Insight: the elfutils package is a great start, as it contains most utilities to perform analysis.

elfkickers

- /usr/bin/ebfc - compiler for [Brainfuck](#)  programming language
- /usr/bin/elfls - shows program headers and section headers with flags
- /usr/bin/elftoc - converts a binary into a C program
- /usr/bin/infect - tool to inject a dropper, which creates setuid file in /tmp
- /usr/bin/objres - creates an object from ordinary or binary data

- /usr/bin/rebind - changes bindings/visibility of symbols in ELF file
- /usr/bin/sstrip - strips unneeded components from ELF file

Insight: the author of the ELFKickers package focuses on manipulation of ELF files, which might be great to learn more when you find malformed ELF binaries.

pax-utils

- /usr/bin/dumpelf - dump internal ELF structure
- /usr/bin/lddtree - like ldd, with levels to show dependencies
- /usr/bin/pspax - list ELF/PaX information about running processes
- /usr/bin/scanelf - wide range of information, including PaX details
- /usr/bin/scanmacho - shows details for Mach-O binaries (Mac OS X)
- /usr/bin/symtree - displays a leveled output for symbols

Notes: Several of the utilities in this package can scan recursively in a whole directory. Ideal for mass-analysis of a directory. The focus of the tools is to gather PaX details. Besides ELF support, some details regarding Mach-O binaries can be extracted as well.

Example output:

```
scanelf -a /bin/ps
TYPE      PAX    PERM ENDIAN STK/REL/PTL TEXTREL RPATH BIND FILE
ET_EXEC PeMRxS 0755 LE RW-  R-- RW-      -      -      LAZY /bin/ps
```

prelink

- /usr/bin/execstack - display or change if stack is executable
- /usr/bin/prelink - remaps/relocates calls in ELF files, to speed up the process

Example binary file

If you want to create a binary yourself, simply create a small C program, and compile it. Here is an example, which opens /tmp/test.txt, reads the contents into a buffer and displays it. Make sure to create the related /tmp/test.txt file.

```
#include <stdio.h>;

int main(int argc, char **argv)
{
    FILE *fp;
    char buff[255];

    fp = fopen("/tmp/test.txt", "r");
    fgets(buff, 255, fp);
    printf("%s\n", buff);
    fclose(fp);

    return 0;
}
```

This program can be compiled with the gcc command

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

Frequently Asked Questions

What is ABI?

ABI is short for Application Binary Interface and specifies a low-level interface between the operating system and a piece of executable code.

What is ELF?

ELF is short for Executable and Linkable Format. It is a formal specification that defines how instructions are stored in executable code.

How can I see the file type of an unknown file?

Use the `file` command to do the first round of analysis. This command may be able to show the details based on header information or magic data.

Conclusion

ELF files are for execution or for linking. Depending on the primary goal, it contains the required segments or sections. Segments are viewed by the kernel and mapped into memory (using `mmap`). Sections are viewed by the linker to create executable code or shared objects.

The ELF file type is very flexible and provides support for multiple CPU types, machine architectures, and operating systems. It is also very extensible: each file is differently constructed, depending on the required parts.

Headers form an important part of the file, describing exactly the contents of an ELF file. By using the right tools, you can gain a basic understanding of the purpose of the file. From there on, you can further inspect the binaries. This can be done by determining the related functions it uses or strings stored in the file. A great start for those who are into malware research, or want to know better how processes behave (or not behave!).

More resources

If you like to know more about ELF and reverse engineering, you might like the work we are doing at Linux Security Expert. Part of a training program, we have a [reverse engineering module](#) with practical lab tasks.

For those who like reading, a good in-depth document: [ELF Format](#) and the [ELF document](#) authored by Brian Raiter (ELFkickers).

Tip: If you like to get better in the analyzing files and samples, then start using the popular [binary analysis tools](#) that are available.

Was this article useful to you? Become part of our community and help others by sharing the article with your favorite website or on social media. Any questions or feedback? Let it [know](#)