ELF

From OSDev Wiki

ELF (Executable and Linkable Format) was designed by Unix System Laboratories while working with Sun Microsystems on SVR4 (UNIX System V Release 4.0). Consequently, ELF first appeared in Solaris 2.0 (aka SunOS 5.0), which is based on SVR4. The format is specified in the System V ABI.

A very versatile file format, it was later picked up by many other operating systems for use as both executable files and as shared library files. It does distinguish between TEXT, DATA and BSS.

Today, ELF is considered the standard format on Unix-alike systems. While it has some drawbacks (e.g., using up one of the scarce general purpose registers of the IA32 when using position-independent code), it is well supported and documented.

Executable Formats

Microsoft

16 bit:
COM
MZ
NE
32/64 bit:
PE
Mixed (16/32 bit):
LE

***nix**

A.out
ELF

Contents

File Structure

ELF is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking. An ELF file is divided into sections. For an executable program, these are the text section for the code, the data section for global variables and the rodata section that usually contains constant strings. The ELF file contains headers that describe how these sections should be stored in memory.

Note that depending on whether your file is a linkable or an executable file, the headers in the ELF file won't be the same: process.o, result of gcc -c process.c $SOME_FLAGS

```
C32/kernel/bin/.process.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000333  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000050  00000000  00000000  00000380  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000003d0  2**2
                  ALLOC
  3 .note         00000014  00000000  00000000  000003d0  2**0
                  CONTENTS, READONLY
  4 .stab         000020e8  00000000  00000000  000003e4  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .stabstr      00008f17  00000000  00000000  000024cc  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .rodata       000001e4  00000000  00000000  0000b400  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .comment      00000023  00000000  00000000  0000b5e4  2**0
                  CONTENTS, READONLY
```

The 'flags' will tell you what's actually available in the ELF file. Here, we have symbol tables and relocation: all that we need to link the file against another, but virtually no information about how to load the file in memory (even if that could be guessed). We don't have the program entry point, for instance, and we have a sections table rather than a program header.

.text	where code stands, as said above. objdump -drS .process.o will show you that
.data	where global tables, variables, etc. stand. objdump -s -j .data .process.o will hexdump it.
.bss	don't look for bits of .bss in your file: there's none. That's where your uninitialized arrays and variable are, and the loader 'knows' they should be filled with zeroes ... there's no point storing more zeroes on your disk than there already are, is it ?
.rodata	that's where your strings go, usually the things you forgot when linking and that cause your kernel not to work. objdump -s -j .rodata .process.o will hexdump it. Note that depending on the compiler, you may have more sections like this.
.comment & .note	just comments put there by the compiler/linker toolchain

.stab & .stabstr	debugging symbols & similar information.

/bin/bash, a real executable file

```
/bin/bash:     file format elf32-i386
/bin/bash
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08056c40

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000e0 memsz 0x000000e0 flags r-x
```

The program header itself... taking 224 bytes, and starting at offset 0x34 in the file

```
  INTERP off    0x00000114 vaddr 0x08048114 paddr 0x08048114 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
```

The program that should be used to 'execute' the binary. Here, it reads as '/lib/ld-linux.so.2', which means some dynamic libraries linking will be required before we run the program.

```
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0007411c memsz 0x0007411c flags r-x
```

Now we're requested to read 7411c bytes, starting at file's start (?) and being 7411c bytes large (that's virtually the whole file!), which will be read-only but executable. They'll be to appear starting at virtual address 0x08048000 for the program to work properly.

```
    LOAD off    0x00074120 vaddr 0x080bd120 paddr 0x080bd120 align 2**12
         filesz 0x000022ac memsz 0x000082d0 flags rw-
```

More bits to load, (likely to be .data section). Notice that the 'filesize' and 'memsize' differ, which means the .bss section will actually be allocated through this statement, but left as zeroes while 'real' data only occupy first 0x22ac bytes starting at virtual address 0x80bd120.

```
 DYNAMIC off    0x00075f4c vaddr 0x080bef4c paddr 0x080bef4c align 2**2
         filesz 0x000000e8 memsz 0x000000e8 flags rw-
```

The dynamic sections are used to store information used in the dynamic linking process, such as required libraries and relocation entries.

```
   NOTE off    0x00000128 vaddr 0x08048128 paddr 0x08048128 align 2**2
        filesz 0x00000020 memsz 0x00000020 flags r--
```

NOTE sections contain information left by either the programmer or the linker, for most programs linked using the GNU `ld` linker it just says 'GNU'

```
EH_FRAME off    0x000740f0 vaddr 0x080bc0f0 paddr 0x080bc0f0 align 2**2
         filesz 0x0000002c memsz 0x0000002c flags r--
```

that's for Exception Handler information, in case we should link against some C++ binaries at execution (afaik).

```
/bin/bash, loaded (as in /proc/xxxx/maps)

08048000-080bd000 r-xp 00000000 03:06 30574      /bin/bash
080bd000-080c0000 rw-p 00074000 03:06 30574      /bin/bash
080c0000-08103000 rwxp 00000000 00:00 0
40000000-40014000 r-xp 00000000 03:06 27304      /lib/ld-2.3.2.so
40014000-40015000 rw-p 00013000 03:06 27304      /lib/ld-2.3.2.so
```

We can recognize our 'code bits' and 'data bits', by stating that the second one should be loaded at 0x080bd*120* and that it starts in file at 0x00074*120*, we actually preserved page-to-disk blocks mapping (e.g. if page 0x80bc000 is missing, just fetch file blocks from 0x75000). That means, however, that a part of the code is mapped twice, but with different permissions. I suggest you do give them different physical pages too if you don't want to end up with modifiable code.

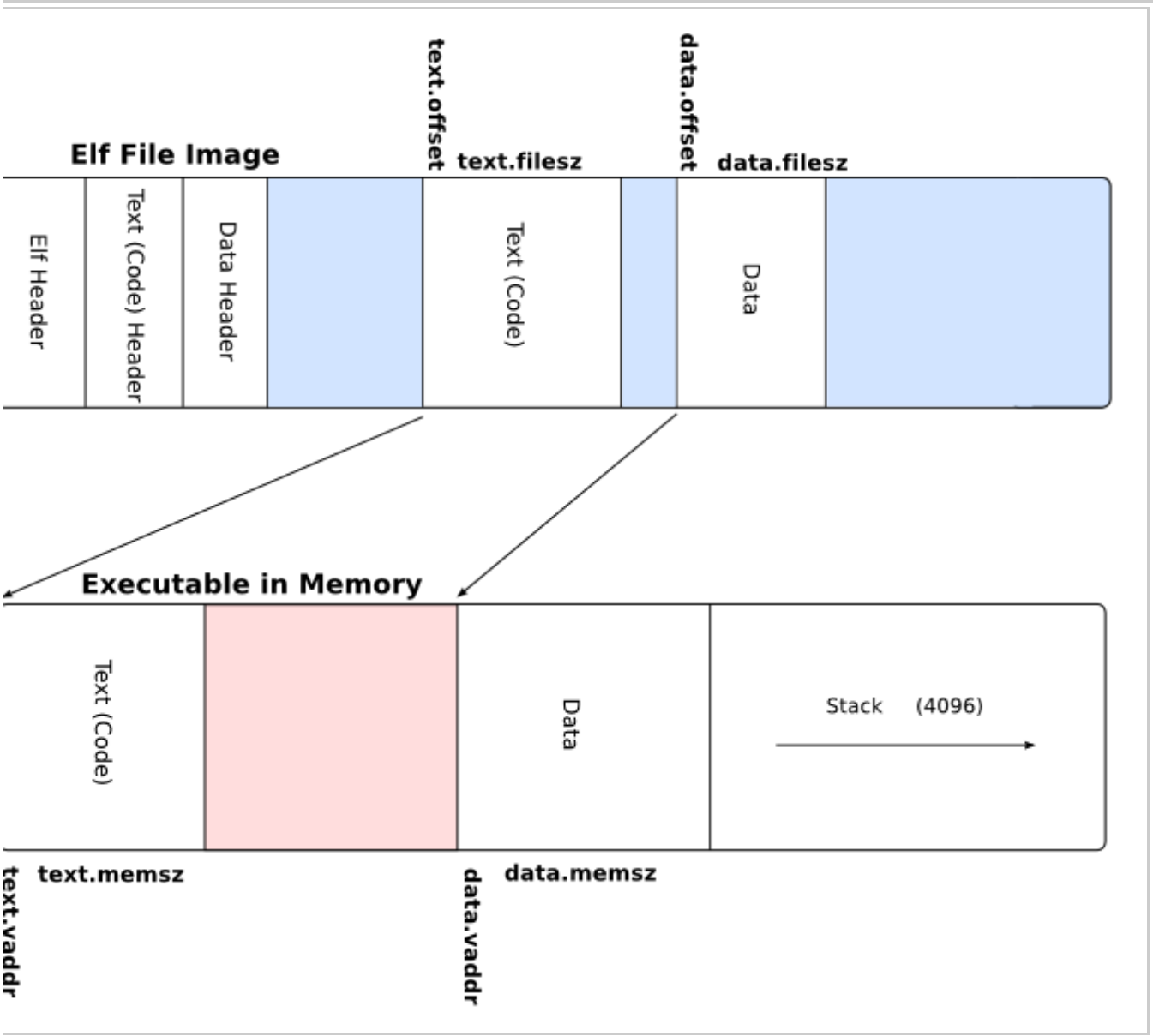
Loading ELF Binaries

The ELF file format is described in the ELF Specification. The most relevant sections for this project are 1.1 to 1.4 and 2.1 to 2.7.

The steps involved in identifying the sections of the ELF file are:

- Read the ELF Header. The ELF header will always be at the very beginning of an ELF file. The ELF header contains information about how the rest of the file is laid out. You are interested only in the program headers.
- Find the Program Headers, which specify where in the file to find the text and data sections and where they should end up in the executable image.

There are a few simplifying assumptions you can make about the types and location of program headers. In the files you will be working with, there will always be one text header and one data header. The text header will be the first program header and the data header will be the second program header. This is not generally true of ELF files, but it will be true of the programs you will be responsible for.



cutable image and elf binary can being mapped onto each other

The file geekos/include/geekos/elf.h provides data types for structures which match the format of the ELF and program headers.

This is a rough guideline for what Parse_ELF_Executable() has to do:

1. Check that exeFileData is non-null and exeFileLength is large enough to accommodate the ELF headers and phnum program headers.
2. Check that the file starts with the ELF magic number (4 bytes) as described in figure 1-4 (and subsequent table) on page 11 in the ELF specification.
3. Check that the ELF file has no more than EXE_MAX_SEGMENTS program headers (phnum field of the elfHeader).

4. Fill in numSegments and entryAddr fields of the exeFormat output variable.
5. For each program header k in turn, fill in the corresponding segmentList[k] array element of exeFormat with offsetInFile, lengthInFile, startAddress, sizeInMemory, protFlags with information from that program header k. See figure 2-1 on page 33 in the ELF specification.

Relocation

Relocation becomes handy when you need to load, for example, modules or drivers. It's possible to use the "-r" option to ld to permit you to have multiple object files linked into one big one, which means easier coding and faster testing.

The basic outline of things you need to do for relocation:

1. Check the object file header (it has to be ELF, not PE, for example)
2. Get a load address (eg. all drivers start at 0xA0000000, need some method of keeping track of driver locations)
3. Allocate enough space for all program sections (ST_PROGBITS)
4. Copy from the image in RAM to the allocated space
5. Go through all sections resolving external references against the kernel symbol table
6. If all succeeded, you can use the "e_entry" field of the header as the offset from the load address to call the entry point (if one was specified), or do a symbol lookup, or just return a success error code.

Once you can relocate ELF objects you'll be able to have drivers loaded when needed instead of at startup - which is always a Good Thing (tm).

See Also

Articles

- System V ABI
- Modular Kernel

External Links

- The ELF file format (http://www.skyfree.org/linux/references/ELF_Format.pdf) in detail
- ELF Format Specifications (http://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-46512.html#scrolltoc) Detailed and up-to-date ELF information (including SPARC in depth) by Oracle.
- System V ABI (http://www.sco.com/developers/gabi/latest/contents.html) about ELF

- LSB specifications (http://www.linuxfoundation.org/en/Specifications) See (generic or platform-specific) 'Core' specifications for additional ELF information.
- Executable and Linkable Format on Wikipedia (http://en.wikipedia.org/wiki/Executable_and_Linkable_Format) ,which contains a detail of elf references
- The ELF file format(64-bit) (http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf) ELF 64-Bit, General extension to ELF32.
- x86-64 ABI (http://www.x86-64.org/documentation/abi.pdf) Documented x86-64 specific extensions with ELF64.
- Manually Creating an ELF Executable (http://www.robinhoksbergen.com/papers/howto_elf.html) Detailed guide on how to create ELF binaries from scratch.

Retrieved from "http://wiki.osdev.org/index.php?title=ELF&oldid=15942"
Categories: ABI | Executable Formats | Standards

- This page was last modified on 8 April 2014, at 13:09.
- This page has been accessed 113,532 times.