Introduction to ELF

Tools, Red Hat, Inc.
Marek Polacek
polacek@redhat.com

Contents

1 **General Info**

- File Types
- ELF Header
- Segments
- Sections

2 **Something about symbols**

- Symbol Binding
- Symbol Visibility

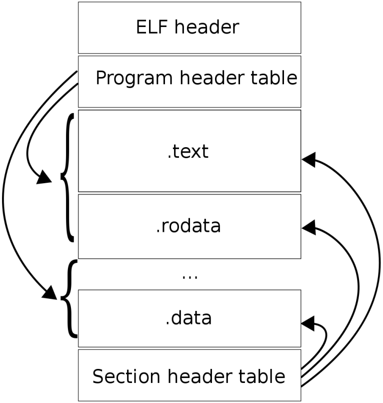
Section 1

General Info

General Info

- ELF == Executable and Linkable Format
- standard file format for executables, object code, shared libraries, and core dumps
- defined by gABI and psABI
- there are other formats as well: a.out, COFF, PE, Mach-O, COM, ...
- dual nature: an ELF file is a set of segments and sections
    - kernel sees segments, maps them into virtual address space using `mmap(2)` syscall
    - linker sees sections, combines them into executable/shared object
- in the kernel: see `fs/binfmt_elf.c`

ELF File Format



ELF File Types

- executables (ET_EXEC)
  - runnable program, must have segments
- object file (ET_REL, *.o)
  - links with other object files, must have sections
- dynamic libraries (ET_DYN, *.so)
  - links with other object files/executables
  - has both segments and sections
- core files (ET_CORE)
  - generated e.g. when program receives SIGABRT et al
  - has no sections, has segments (PT_LOAD/PT_NOTE)
- example question: and what about static libraries?

redhat.

ELF Header

- starts always at the beginning of the file
- defined in `Elf64_Ehdr` structure:

e_ident magic bytes (`0x7fELF`), class, ABI version, ...

e_type object file type—`ET_{REL,DYN,EXEC,CORE}`

e_machine required architecture—`EM_X86_64`, ...

e_version `EV_CURRENT`, always ”1”

e_entry virt. addr. of entry point, `_dl_start, jmp *%r12`

e_phoff program header offset

e_shoff section header offset

e_flags CPU-specific flags

e_ehsize ELF header size

e_phentsize size of program header entry, consistency check

ELF Header

e_phnum number of program header entries

e_shentsize size of section header entry

e_shnum number of section header entries

e_shstrndx section header string table index

```
$ readelf -Wh /lib64/ld-linux-x86-64.so.2
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:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x37e6c016e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          166656 (bytes into file)
  ...
```

ELF Header—an example

```
#include <elf.h>
bool
is_elf_p (const char *fname)
{
  int fd = open64 (fname, O_RDONLY);
  if (fd == -1)
    goto out;
  char ident[EI_NIDENT];
  if (pread64 (fd, ident, EI_NIDENT, 0) != EI_NIDENT)
    goto out;
  return memcmp (&ident[EI_MAG0], ELFMAG, SELFMAG) != 0;
  out:
    /* ...  */
    return false;
}
```

Program Header

- an array of structures, each describing a segment
- segments contain sections
- defined in `Elf64_Phdr` structure:

p_type	segment type, described later
p_flags	segment flags—`PF_R, PF_W, PF_X`
p_offset	segment file offset from beginning of the file
p_vaddr	segment virtual address
p_paddr	segment physical address
p_memsz	segment size in memory
p_filesz	segment size in file
p_align	segment alignment

Segment Types

PT_NULL array element is unused

PT_LOAD loadable entry in the segment table, OS/rtld loads all segments of this type, we can have more than one, sorted by `p_vaddr`

PT_DYNAMIC dynamic linking information

PT_INTERP path to the dynamic linker, in an executable; see `$ readelf -Wp .interp <foo>`

PT_NOTE OS/ABI requirements, e.g. min. kernel version

PT_SHLIB who knows; ignored

PT_PHDR address and size of the segment table

PT_TLS Thread-Local Storage template

Segment Types

GNU extensions:

PT_GNU_EH_FRAME sorted table of unwind information. GCC uses this table to find the appropriate handler for an exception.

PT_GNU_STACK whether we need an executable stack; permission of the stack in memory

PT_GNU_RELRO which part of the memory should be read-only after applying dynamic relocations

PT_GNU_HEAP so far only Gentoo uses this

- example question: can the segments overlap?
- yes, and they often do: see `PT_INTERP` and `PT_LOAD`, for instance

Segment Types

GNU extensions:

PT_GNU_EH_FRAME sorted table of unwind information. GCC uses this table to find the appropriate handler for an exception.

PT_GNU_STACK whether we need an executable stack; permission of the stack in memory

PT_GNU_RELRO which part of the memory should be read-only after applying dynamic relocations

PT_GNU_HEAP so far only Gentoo uses this

- example question: can the segments overlap?
- yes, and they often do: see `PT_INTERP` and `PT_LOAD`, for instance

Segments Example

```
$ readelf -Wl /lib64/ld-linux-x86-64.so.2
Elf file type is DYN (Shared object file)
Entry point 0x37e6c016e0
There are 7 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x000000 0x00000037e6c00000 0x00000037e6c00000 0x021a30 0x021a30 R E 0x200000
  LOAD           0x021b30 0x00000037e6e21b30 0x00000037e6e21b30 0x0014c8 0x001758 RW  0x200000
  DYNAMIC        0x021de8 0x00000037e6e21de8 0x00000037e6e21de8 0x0001b0 0x0001b0 RW  0x8
  NOTE           0x0001c8 0x00000037e6c001c8 0x00000037e6c001c8 0x000024 0x000024 R   0x4
  GNU_EH_FRAME   0x01f164 0x00000037e6c1f164 0x00000037e6c1f164 0x000664 0x000664 R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x8
  GNU_RELRO      0x021b30 0x00000037e6e21b30 0x00000037e6e21b30 0x0004d0 0x0004d0 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_d
          .rela.dyn .rela.plt .plt .text .rodata .stapsdt.base .eh_frame_hdr .eh_frame
   01     .init_array .data.rel.ro .dynamic .got .data .bss
   02     .dynamic
   03     .note.gnu.build-id
   04     .eh_frame_hdr
   05
   06     .init_array .data.rel.ro .dynamic .got
```

redhat

Section Header

- an array of structures, each describing a section
- defined in `Elf64_Shdr` structure:

sh_name name (string table index)

sh_type section type, described later

sh_flags section flags— `SHF_{WRITE,ALLOC,EXECINSTR,MERGE,STRINGS,...}`

sh_offset offset from the beginning of the file to the first byte in the section

sh_addr virt. addr. of the section, 0 in `ET_REL`

sh_size section's size in bytes

sh_link section header table index link, depends on `sh_type`

sh_info extra information, depends on the `sh_type`

sh_addralign section alignment

sh_entsize entry size if section contains a table

Section Types

There are many of them, we mention only some:

SHT_PROGBITS bits of the program

SHT_SYMTAB symbol table; an array of ELF symbol structures

SHT_STRTAB string table; holds null-terminated strings

SHT_RELA relocation table

SHT_HASH hash table used by rtld to speed symbol lookup

SHT_DYNAMIC dynamic tags used by rtld, same as `PT_DYNAMIC`

SHT_NOBITS zero-initialized data

Sections Example

```
$ readelf -WS x.o
There are 16 section headers, starting at offset 0x288:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 000000 00  AX  0   0  4
  [ 2] .data             PROGBITS        0000000000000000 000040 000000 00  WA  0   0  4
  [ 3] .bss              NOBITS          0000000000000000 000040 000000 00  WA  0   0  4
  [ 4] .rodata.str1.1    PROGBITS        0000000000000000 000040 000011 01 AMS  0   0  1
  [ 5] .text.startup     PROGBITS        0000000000000000 000060 0000e3 00  AX  0   0 16
  [ 6] .rela.text.startup RELA            0000000000000000 000828 0003c0 18     14   5  8
  [ 7] .ctors            PROGBITS        0000000000000000 000148 000018 00  WA  0   0  8
  [ 8] .rela.ctors       RELA            0000000000000000 000be8 000048 18     14   7  8
  [ 9] .comment          PROGBITS        0000000000000000 000160 00002d 01  MS  0   0  1
  [10] .note.GNU-stack   PROGBITS        0000000000000000 00018d 000000 00      0   0  1
  [11] .eh_frame         PROGBITS        0000000000000000 000190 000070 00   A  0   0  8
  [12] .rela.eh_frame    RELA            0000000000000000 000c30 000060 18     14  11  8
  [13] .shstrtab         STRTAB          0000000000000000 000200 000082 00      0   0  1
  [14] .symtab           SYMTAB          0000000000000000 000688 000180 18     15  14  8
  [15] .strtab           STRTAB          0000000000000000 000808 00001e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Special Sections

There are many of them, we mention only some:

.text	executable instructions
.bss/.tbss	Block Started by Symbol, uninitialized data, zeroes
.data/.tdata	initialized data/`__thread` data
.rodata	read-only data
.dynamic	dynamic linking information—`DT_{NEEDED,RUNPATH,SONAME,...}`
.got{,.plt}	Global Offset Table
.plt	Procedure Linkage Table
.gnu.hash	symbol hash table
.strtab	string table
.init/.fini	executable insns, initialization code
.{init,fini}_array	array of function pointers to init functions

Section 2

Something about symbols

Symbol Binding

There are three most basic types of binding:

STB_LOCAL not visible outside the object file, `static`

STB_GLOBAL visible to all object files being combined

STB_WEAK can be overriden by stronger definition, example follows

- see `weak_alias` and `strong_alias` macros in glibc

STB_WEAK—an example

main.c

```
extern void foo (void);
int
main (void)
{
  foo ();
}
```

foo.c

```
#include <stdio.h>
void
foo (void)
{
  puts (__FILE__);
}
```

foo2.c

```
#include <stdio.h>
void
foo (void)
{
  puts (__FILE__);
}
```

```
$ gcc main.c foo.c foo2.c
/tmp/ccGD9LA8.o: In function
'foo': foo2.c:(.text+0x0):
multiple definition of 'foo'
/tmp/cc1gCusT.o:foo.c:(.text+0x0):
first defined here collect2: ld
returned 1 exit status
```

STB_WEAK—an example

main.c

```
extern void foo (void);
int
main (void)
{
  foo ();
}
```

foo.c

```
#include <stdio.h>
void __attribute__ ((weak))
foo (void)
{
  puts (__FILE__);
}
```

foo2.c

```
#include <stdio.h>
void
foo (void)
{
  puts (__FILE__);
}
```

```
$ gcc main.c foo.c foo2.c
$ ./a.out
foo2.c
```

Symbol Visibility

STV_DEFAULT default symbol visibility rules; symbol is exported and can be interposed

STV_HIDDEN symbol is unavailable outside the library

STV_PROTECTED not preemptible, not exported; never use this

STV_INTERNAL processor specific hidden class

GCC Support

GCC supports setting global visibility:

-fvisibility=default all symbols are `STV_DEFAULT` by default

-fvisibility=hidden all symbols are `STV_HIDDEN` by default

...and per-symbol visibility:

```
long int def __attribute__ ((visibility ("default")));
long int hid __attribute__ ((visibility ("hidden")));
```

or:

```
#pragma GCC visibility push(hidden)
int hid1;
int hid2;
#pragma GCC visibility pop
```

Conclusion

- slides are available at:

`http://people.redhat.com/mpolacek/src/devconf2012.pdf`

The end.

Thanks for listening.