Software Security

Objdump (ELF) Binaries

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2022 / 2023

Part I

Introduction

Three Kinds of Compiled Files (Modules)

- Relocatable object file (.o file)
  - Contains code (machine instructions) and data (constants and variables) in a form that can be combined with other relocatable object files to form executable file.
  - Each .o file is produced from exactly one source .c file

- Executable object file (a.out file)
  - Contains code and data in a form that can be copied directly into memory and then executed.

- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLL) by Windows.

Executable and Linkable Format (ELF)

- Standard binary format for object files.
- Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux.

- One unified format for
  - Relocatable object files .o
  - Executable object files a.out
  - Shared object file .so

- Generic name: ELF binaries (only study 64-bit ELF binaries)

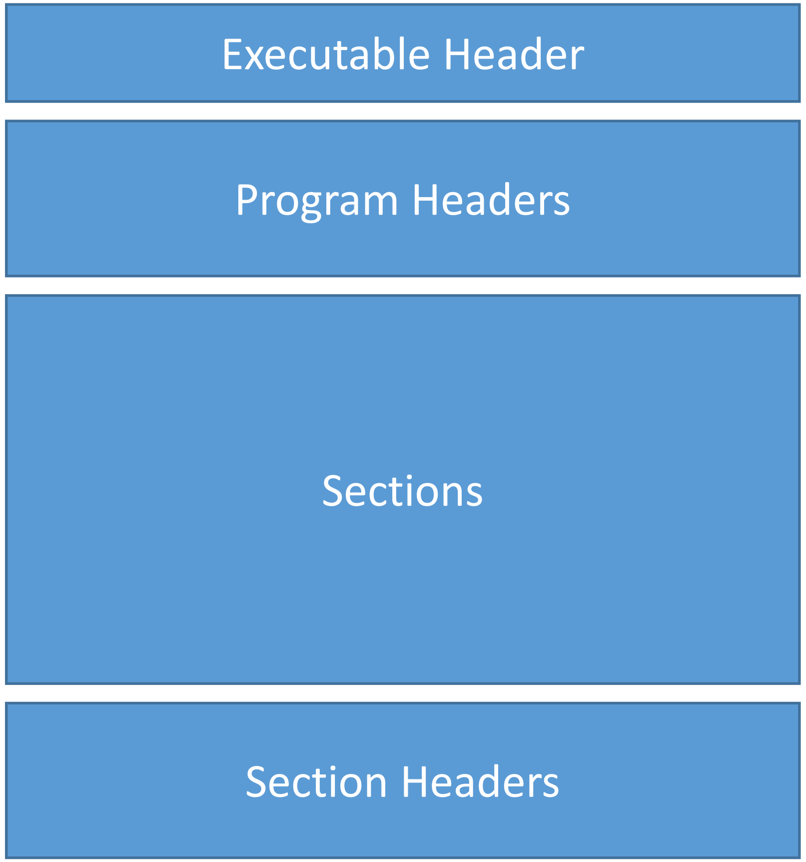
ELF Views

- ELF describes two separate ‘views’ of an executable:
  - A linking view
  - A loading view

- Linking view is used at static link time to combine relocatable files.
  - Section Headers are used during compile-time linking;
  - It tells the link editor ld how to resolve symbols, and how to combine several ELF objects into one executable.

- Loading view is used at run time to load and execute program.
  - Segment Headers are used during execution;
  - It tells the runtime linker ld.so what to load into memory and how to find dynamic linking information.

64-bit ELF Binary



ELF Components

- Executable Header
  - Tells you that what kind of an ELF file is.

- A series of (optional) program headers (or segments)
  - Provides the execution view.

- A number of sections
  - The code and data of programs.

- A series of (optional) section headers
  - Each denoting the property of its related section.

Section vs Segment

- Section: exists before linking, in object files.
  - Sections contain raw data to be loaded into memory.
  - Sections also contain metadata that will disappear at runtime.

- Segment exists after linking, in the executable files.
  - One or more sections will be put inside a single segment by the linker.
  - Segments contain information about how each section should be loaded into memory by the OS, notably location and permissions.

Part II

Executable (ELF) Header

ELF64_Ehdr in /usr/include/elf.h

```c
typedef struct {
    unsigned char e_ident[16]; /* Magic number and other info       */
    uint16_t      e_type;      /* Object file type                  */
    uint16_t      e_machine;   /* Architecture                      */
    uint16_t      e_version;   /* Object file version               */
    uint16_t      e_entry;     /* Entry point virtual address       */
    uint16_t      e_phoff;     /* Program header offset             */
    uint16_t      e_shoff;     /* Section header offset             */
    uint16_t      e_flags;     /* Processor-specifc flags           */
    uint16_t      e_ehsize;    /* ELF header size in bytes          */
    uint16_t      e_phentsize; /* Program header size               */
    uint16_t      e_phnum;     /* Program header count              */
    uint16_t      e_shentsize; /* Section header size               */
    uint16_t      e_shnum;     /* Section header count              */
    uint16_t      e_shstrndx;  /* Section header string table index */
} ELF64_Ehdr;
```

E_ident Array

- 4-byte magic value:
  - 7F 45 4C 46
  - Hexadecimal 0x7F followed by the ASCII character code for the letters ‘E’, ‘L’ and ‘F’.
- EI_CLASS
  - Denotes whether the binary is for 32-bit or 64-bit architecture.
- EI_DATA
  - Indicates the endianness of the binary.
- EI_VERSION
  - Version of the ELF specification.
  - The only valid value is 1.
- EI_OSABI
  - Non zero value indicates the use of OS-specific extensions.
- EI_ABIVERSION
  - Often set to zero.
- 7-byte EI_PAD
  - Reserved for future use.
  - Currently set to zero.

ELF Header Example 1/4

```
sabtmoha@sabtmoha:~/sw_secu$ readelf -h main
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
```

The fields e_type, e_machine, and e_version

- E_type:
  - ET_REL: relocatable object file.
  - ET_EXEC: executable binary.
  - ET_DYN: shared object file.

- E_machine
  - Denotes the CPU architecture.
  - EM_X86_64.
  - EM_386.
  - EM_ARM.

- E_version
  - Similar to EI_VERSION in the e_ident array.

E_entry

- Denotes the entry point of the binary.
  - This is the virtual address at which the execution would start.

ELF Header Example 2/4

```
sabtmoha@sabtmoha:~/sw_secu$ readelf -h main
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:               0x1080
```

The fields e_phoff and e_shoff

- The only data structure that can be assumed at a fixed location is the executable header, which is always at the beginning.

- E_phoff
  - Indicates the file offset to the beginning of the program header

- E_shoff
  - Indicates the file offset to the beginning of the section header

- File offsets are NOT virtual addresses ; they mean the number of bytes to read to get to the headers.

E_flag

- Provides architecture specific flags.
- For x86 binaries, e_flags is typically set to zero.

ELF Header Example 3/4

```
sabtmoha@sabtmoha:~/sw_secu$ readelf -h main
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:               0x1080
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14752 (bytes into file)
  Flags:                             0x0
```

The fields e_ehsize, e_*entsize and e_*num

- E_ehsize
  - Specifies the size of the executable header.
  - For 64-bit binaries, it is always equal to 64 bytes.

- E_phrntsize and e_shentsize
  - The size of each header.

- E_phnum and e_shnum
  - The number of headers.

ELF Header Example 4/4

```
sabtmoha@sabtmoha:~/sw_secu$ readelf -h main
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:               0x1080
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14752 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
```

E_shstrndx

- The section .shstrtab contains NULL-terminated strings that store the name of all the sections in the binary.

- The field e_shstrndx is the index of the .shstrtab section in the section header table.

Section Names

```
sabtmoha@sabtmohav2:~$ readelf -x .shstrtab main

Hex dump of section '.shstrtab':
  0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strtab
  0x00000010 002e7368 73747274 6162002e 696e7465 ..shstrtab..inte
  0x00000020 7270002e 6e6f7465 2e676e75 2e627569 rp..note.gnu.bui
  0x00000030 6c642d69 64002e6e 6f74652e 4142492d ld-id..note.ABI-
  0x00000040 74616700 2e676e75 2e686173 68002e64 tag..gnu.hash..d
  0x00000050 796e7379 6d002e64 796e7374 72002e67 ynsym..dynstr..g
  0x00000060 6e752e76 65727369 6f6e002e 676e752e nu.version..gnu.
  0x00000070 76657273 696f6e5f 72002e72 656c612e version_r..rela.
  0x00000080 64796e00 2e72656c 612e706c 74002e69 dyn..rela.plt..i
  0x00000090 6e697400 2e706c74 2e676f74 002e7465 nit..plt.got..te
  0x000000a0 7874002e 66696e69 002e726f 64617461 xt..fini..rodata
  0x000000b0 002e6568 5f667261 6d655f68 6472002e ..eh_frame_hdr..
  0x000000c0 65685f66 72616d65 002e696e 69745f61 eh_frame..init_a
  0x000000d0 72726179 002e6669 6e695f61 72726179 rray..fini_array
  0x000000e0 002e6479 6e616d69 63002e67 6f742e70 ..dynamic..got.p
  0x000000f0 6c74002e 64617461 002e6273 73002e63 lt..data..bss..c
  0x00000100 6f6d6d65 6e7400                     omment.
```

Part III

Section Headers

Headers

- Sections do not have any predetermined structure.

- Each section is described by a section header
  - Which denotes the properties of the section;
  - And allows you to locate its content.

- Section Headers are at the end of the ELF file.

ELF64_Shdr in /usr/include/elf.h

```
typedef struct {
    uint32_t  sh_name;      /* section name (string tbl index)   */
    uint32_t  sh_type;      /* section type                      */
    uint64_t  sh_flags;     /* section flags                     */
    uint64_t  sh_addr;      /* section virtual addr at execution */
    uint64_t  sh_offset;    /* section file offset               */
    uint64_t  sh_size;      /* section size in bytes             */
    uint32_t  sh_link;      /* link to another section           */
    uint32_t  sh_info;      /* additional section information    */
    uint64_t  sh_addralign; /* section alignement                */
    uint64_t  sh_entsize;   /* entry size if section holds table */
} ELF64_Shdr;
```

The fields sh_name and sh_type

- Sh_name
  - Index into the string table of the section .shstrtab.
  - This is not for machines, but for humans.

- Sh_type
  - SHT_PROGBITS: contain program data, such as machine instructions.
  - SHT_SYMTAB: static symbol tables.
  - SHT_DYNSYM: symbol tables for the dynamic linker.
  - SHT_STRTAB: string tables.
  - SHT_REL and SHT_RELA: used for static linking purposes.

sh_flags

- SHF_WRITE
  - Indicates whether the section is writable at runtime.

- SHF_ALLOC
  - Contents are to be loaded when executing the program.

- SHF_EXECINSTR
  - Contains executable instructions.

The fields sh_addr, sh_offset and sh_size

- These fields describe the virtual address, file offset and size (in bytes) of the section.

- Exercise: why sh_addr in the section view?

The fields sh_addralign and sh_entsize

- Sh_addralign
  - The required alignment in memory for faster execution.

- Sh_entsize
  - Some sections, such as symbol tables and relocation tables, contain a table of well-defined data structures.
  - This field indicates the size of each entry of this table.

Section Headers Example

```
sabtmoha@sabtmoha:~/sw_secu$ readelf -S main.o
There are 14 section headers, starting at offset 0x348:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000027  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000270
       0000000000000048  0000000000000018   I      11     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000067
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  00000067
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  00000067
       0000000000000004  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  0000006b
       000000000000002b  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000096
       0000000000000000  0000000000000000           0     0     1
  [ 8] .note.gnu.propert NOTE             0000000000000000  00000098
       0000000000000020  0000000000000000   A       0     0     8
  [ 9] .eh_frame         PROGBITS         0000000000000000  000000b8
       0000000000000038  0000000000000000   A       0     0     8
  [10] .rela.eh_frame    RELA             0000000000000000  000002b8
       0000000000000018  0000000000000018   I      11     9     8
  [11] .symtab           SYMTAB           0000000000000000  000000f0
       0000000000000150  0000000000000018          12    10     8
  [12] .strtab           STRTAB           0000000000000000  00000240
       000000000000002f  0000000000000000           0     0     1
  [13] .shstrtab         STRTAB           0000000000000000  000002d0
       0000000000000074  0000000000000000           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),
  l (large), p (processor specific)
```

Stripping Section Table

- Find where section table is
  - readelf –h

- Remove it
  - truncate -s

- Verify that you can run the file
- Try to use readelf to display some sections.

Part IV

Sections

Sections View

- Typical ELF files that you'll find on a GNU/Linux system are organized into a series of standard (orde facto standard) sections.
- For each section, readelf shows the relevant basic information, including the index (in the section header table), name, and type of the section.
- Moreover, you can also see the virtual address, file offset, and size in bytes of the section.
- Finally, readelf also shows the relevant flags for each section, as well as other additional information.

Example of Sections

```
sabtmoha@sabtmohav2:~$ readelf --sections --wide main
There are 30 section headers, starting at offset 0x3a18:

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        00000000000002a8 0002a8 00001c 00   A  0   0  1
  [ 2] .note.gnu.build-id NOTE            00000000000002c4 0002c4 000024 00   A  0   0  4
  [ 3] .note.ABI-tag     NOTE            00000000000002e8 0002e8 000020 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0000000000000308 000308 000024 00   A  5   0  8
  [ 5] .dynsym           DYNSYM          0000000000000330 000330 0000f0 18   A  6   1  8
  [ 6] .dynstr           STRTAB          0000000000000420 000420 000095 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          00000000000004b6 0004b6 000014 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         00000000000004d0 0004d0 000020 00   A  6   1  8
  [ 9] .rela.dyn         RELA            00000000000004f0 0004f0 0000c0 18   A  5   0  8
  [10] .rela.plt         RELA            00000000000005b0 0005b0 000060 18  AI  5  23  8
  [11] .init             PROGBITS        0000000000001000 001000 000017 00  AX  0   0  4
  [12] .plt              PROGBITS        0000000000001020 001020 000050 10  AX  0   0 16
  [13] .plt.got          PROGBITS        0000000000001070 001070 000008 08  AX  0   0  8
  [14] .text             PROGBITS        0000000000001080 001080 0001e1 00  AX  0   0 16
  [15] .fini             PROGBITS        0000000000001264 001264 000009 00  AX  0   0  4
  [16] .rodata           PROGBITS        0000000000002000 002000 000019 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        000000000000201c 00201c 000044 00   A  0   0  4
  [18] .eh_frame         PROGBITS        0000000000002060 002060 000130 00   A  0   0  8
  [19] .init_array       INIT_ARRAY      0000000000003de8 002de8 000008 08  WA  0   0  8
  [20] .fini_array       FINI_ARRAY      0000000000003df0 002df0 000008 08  WA  0   0  8
  [21] .dynamic          DYNAMIC         0000000000003df8 002df8 0001e0 10  WA  6   0  8
  [22] .got              PROGBITS        0000000000003fd8 002fd8 000028 08  WA  0   0  8
  [23] .got.plt          PROGBITS        0000000000004000 003000 000038 08  WA  0   0  8
  [24] .data             PROGBITS        0000000000004038 003038 000010 00  WA  0   0  8
  [25] .bss              NOBITS          0000000000004048 003048 000008 00  WA  0   0  1
  [26] .comment          PROGBITS        0000000000000000 003048 000027 01  MS  0   0  1
  [27] .symtab           SYMTAB          0000000000000000 003070 000660 18     28  45  8
  [28] .strtab           STRTAB          0000000000000000 0036d0 00023b 00      0   0  1
  [29] .shstrtab         STRTAB          0000000000000000 00390b 000107 00      0   0  1
```

The .init and .fini Sections

- The .init section contains executable code that performs initialization tasks and needs to run before any other code in the binary is executed.
  - The system executes the code in the .init section before transferring control to the main entry point of the binary.

- The .fini section is analogous to the .init section, except that it runs after the main program completes, essentially functioning as a kind of destructor

The .init_array and .fini_array Sections

- The .init_array section contains an array of pointers to functions to run when the binary is initialized, before main is called.
  - .init section contains a single startup function that performs some crucial initialization needed to start the executable.
  - .init_array is a data section that can contain as many function pointers as you want, including your own functions.
  - By default, there is an entry for executing frame_dummy.

- As you may have guessed by now, .fini_array is analogous to .init_array, except that .fini_array contains pointers to destructors rather than constructors.
- These sections are convenient places to insert hooks that add initialization or finalization code to the binary to modify its behavior.
  - In gcc, you can mark functions in your C source files as constructors (resp. destructors) by decorating them with __attribute__((constructor)) (resp. __attribute__((destructor))).

The .text Section

- The .text section is where the main code of the program resides.

The .bss, .data and .rodata Sections

- The .rodata section, which stands for “read-only data,” is dedicated to storing constant values.
  - Constant data is usually also kept in its own section to keep the binary neatly organized, though compilers *do* sometimes output constant data in code sections.
- The default values of initialized variables are stored in the .data section, which *is* marked as writable since the values of variables may change at runtime.
- Finally, the .bss section reserves space for uninitialized variables.
  - In the .bss section has type SHT_NOBITS. This is because .bss doesn’t occupy any bytes in the binary as it exists on disk.
  - Variables that live in .bss are zero initialized, and the section is marked as writable.

The .shstrtab, .symtab, .strtab, .dynsym, and .dynstr Sections

- The .shstrtab section is simply an array of NULL-terminated strings that contain the names of all the sections in the binary.
- The .symtab section contains a symbol table that associates a symbolic name with a piece of code or data elsewhere in the binary, such as a function or variable.
- The actual strings containing the symbolic names are located in the .strtab section.
  - Stripped binaries mean that the .symtab and .strtab tables are removed.
- The .dynsym and .dynstr sections are analogous to .symtab and .strtab, except that they contain symbols and strings needed for dynamic linking rather than static linking.

Part V

Program Headers

Introduction

- The *program header table* provides a *segment view* of the binary, as opposed to the *section view* provided by the section header table.

- An ELF segment encompasses zero or more sections, essentially bundling these into a single chunk.

- Since segments provide an execution view, they are needed only for executable ELF files and not for nonexecutable files such as relocatable objects.

ELF64_Phdr in /usr/include/elf.h

```c
typedef struct {
    uint32_t p_type;   /* Segment type             */
    uint32_t p_flags;  /* Segment flags            */
    uint64_t p_offset; /* Segment file offset      */
    uint64_t p_vaddr;  /* Segment virtual address  */
    uint64_t p_paddr;  /* Segment physical address */
    uint64_t p_filesz; /* Segment size in file     */
    uint64_t p_memsz;  /* Segment size in memory   */
    uint64_t p_align;  /* Segment alignment        */
} ELF64_Phdr;
```

The p_type Field

- The p_type field identifies the type of the segment.

- Important values for this field include
  - PT_LOAD: Segments of type are intended to be loaded into memory
  - PT_DYNAMIC: contains the .dynamic section, which tells the interpreter how to parse and prepare the binary for execution.
  - PT_INTERP: contains the .interp section, which provides the name of the interpreter that is to be used to load the binary.

The p_flags Field

- The flags specify the runtime access permissions for the segment.

- Three important types of flags exist:
  - PF_X,
  - PF_W
  - PF_R.

The p_offset, p_vaddr, p_paddr, p_filesz, and p_memsz Fields

- The p_offset, p_vaddr, and p_filesz fields are analogous to the sh_offset, sh_addr, and sh_size fields in a section header.
  - For loadable segments, p_vaddr must be equal to p_offset, modulo the page size (which is typically 4,096 bytes).

- On modern operating systems such as Linux, the field p_paddr is unused and set to zero since they execute all binaries in virtual memory.

- At first glance, it may not be obvious why there are distinct fields for the file size of the segment (p_filesz) and the size in memory (p_memsz).
  - .bss Section as an example.

The p_align Field

- The p_align field is analogous to the sh_addralign field in a section header. It indicates the required memory alignment (in bytes) for the segment.

- If p_align isn't set to 0 or 1, then its value must be a power of 2.