ELF1 7C Executing Background - ELF Study 1999

Young W. Lim

2020-12-21 Mon

Outline

Based on

"Study of ELF loading and relocs", 1999
`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

ELF header

- defines whether to use 32-bit or 64-bit addresses.
- contains three fields that are affected by this setting and offset other fields that follow them.
  - `e_entry` (entry point)
  - `e_phoff` (program header table offset)
  - `e_shoff` (section header table offset)
- The ELF header is 52 or 64 bytes long for 32-bit and 64-bit binaries respectively.

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF header example

```
$ readelf -h /bin/bash
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x805be30
  Start of program headers:          52 (bytes into file)
  Start of section headers:          675344 (bytes into file)
  Flags:                             0x0
  Size of this header:               52
  Size of program headers:           32
  Number of program headers:         8
  Size of section headers:           40
  Number of section headers:         26
  Section header string table index: 25
```

https://greek0.net/elf.html

ELF header fields

```
typedef struct {
  unsigned char   e_ident[EI_NIDENT];
  Elf32_Half      e_type;
  Elf32_Half      e_machine;
  Elf32_Word      e_version;
  Elf32_Addr      e_entry;
  Elf32_Off       e_phoff;
  Elf32_Off       e_shoff;
  Elf32_Word      e_flags;
  Elf32_Half      e_ehsize;
  Elf32_Half      e_phentsize;
  Elf32_Half      e_phnum;
  Elf32_Half      e_shentsize;
  Elf32_Half      e_shnum;
  Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
// 52 bytes for 32-bit machines
```

```
typedef struct {
  unsigned char   e_ident[EI_NIDENT];
  Elf64_Half      e_type;
  Elf64_Half      e_machine;
  Elf64_Word      e_version;
  Elf64_Addr      e_entry;
  Elf64_Off       e_phoff;
  Elf64_Off       e_shoff;
  Elf64_Word      e_flags;
  Elf64_Half      e_ehsize;
  Elf64_Half      e_phentsize;
  Elf64_Half      e_phnum;
  Elf64_Half      e_shentsize;
  Elf64_Half      e_shnum;
  Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
// 64 bytes for 64-bit machines
```

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF header - e_ident field

```
0x00    4       e_ident[EI_MAG0] through e_ident[EI_MAG3]
0x04    1       e_ident[EI_CLASS]
0x05    1       e_ident[EI_DATA]
0x06    1       e_ident[EI_VERSION]
0x07    1       e_ident[EI_OSABI]
0x08    1       e_ident[EI_ABIVERSION]
0x09    7       e_ident[EI_PAD]
```

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF header - e_entry field

- This is the memory address of the entry point
  from where the process starts executing.
  This field is either 32 or 64 bits long
  depending on the format defined earlier.

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

entry (1)

- an entry point is where control is transferred from the operating system to a computer program, at which place the processor enters a program or a code fragment and execution begins.
- This marks the transition from load time (and dynamic link time, if present) to run time

https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-an

entry (2)

1. In some operating systems or programming languages,
   the initial entry is not part of the program
   but of the runtime library
   - the runtime library initializes the program
   - then the runtime library enters the program.
2. In other cases, the program may call the runtime library
   before doing anything when it is entered for the first time,
   - after the &runtime library returns,
     the actual code of the program begins to execute.

https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any

entry (3)

- entry point is used to point at the location at which the OS loader will start a program
  - for a given binary file (ELFBIN), use `readelf -h ELFBIN` to read the binary's header information (-h):
    - `Entry point address:  0x400a80`
  - after running `objdump` on the binary
    - `0000000000400a80 <_start>:`

https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any

entry (4)

- it is the `_start` function that prepares certain parameters/registers before eventually calling main
  - `400aa4:   callq *0x20851e(%rip)` contains a program code.
  - the `_start` function is usually called after all other sections of the binary have been loaded in memory.
- after the main is done, the `hlt` instruction is executed to terminate the execution in this example.
  - the `hlt` instruction is typically never reached since `__libc_start_main` calls `exit(2)` if `main` returns normall

https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any

stripped executables

- with gcc's -g flag, an executable contains debugging information.
  - for each instruction there is information
    which line of the source code generated it,
    the name of the variables in the source code is retained and
    can be associated to the matching memory at runtime etc.
- strip can remove this debugging information
  and other data included in the executable
  which is not necessary for execution
  in order to reduce the size of the executable.

https://unix.stackexchange.com/questions/2969/what-are-stripped-and-not-stripped-e

gcc -s

- gcc being a compiler/linker, its -s option is something done while linking
- it's not configurable
  - it has a set of information which it removes, no more no less.
- removes the relocation information and the symbol table which is not done by strip
  - Note that, removing relocation information would have some effect on address space layout randomization

https://stackoverflow.com/questions/1349166/what-is-the-difference-between-gcc-s-a

strip

- strip can be run on an object file
  which is already compiled.
- has a variety of command-line options to
  configure which information will be removed.
- For example, -g strips only the debug information
- Note that strip is not a bash command,
  though you may be running it from a bash shell.
- It is a command totally separate from bash,
  part of the GNU binary utilities suite.

https://stackoverflow.com/questions/1349166/what-is-the-difference-between-gcc-s-a

finding main function's entry point (1)

- once a program has been stripped,
  there is no straightforward way to locate the function
  that the symbol main would have otherwise referenced.
- The value of the symbol main is not required
  for program start-up:

https://stackoverflow.com/questions/9885545/how-to-find-the-main-functions-entry-p

finding main function's entry point (2)

- in the ELF format, the start of the program is specified by the `e_entry` field of the ELF file header.
- This field normally points to the C library's initialization code, and not directly to main.
- While the C library's initialization code does call `main()` after it has set up the C run time environment, this call is a normal function call that gets fully resolved at link time

https://stackoverflow.com/questions/9885545/how-to-find-the-main-functions-entry-p

Execution sequence

1. Kernel does permission checks
2. Kernel attempts to determine the internal format.
   It finds out it's ELF and that it's dynamically linked.
3. Kernel decodes the structure of the ELF executable,
   finding the interpreter (`ld-linux.so.2` or something).
   It attempts to load the interpreter,
   which itself is a statically linked ELF executable.
4. The interpreter, in user space, looks for and loads
   the shared object files (extension .so, internal format ELF)
   which are needed by the executable.
   Once they are all loaded and relocated, control is passed
   to the executable itself, at the entry point established.

https://www.quora.com/How-is-a-elf-file-executed-in-Linux

Manual load and execution (1)

1. Read the program headers
   - to find the LOAD directives and
   - determine the total length of mappings in pages.
2. Map the lowest-address LOAD directive
   with the total length (which may be greater than the file length),
   letting mmap assign you an address.
   This will reserve contiguous virtual address space.
3. map the remainin LOAD directives
   over top of parts of this mapping using MAP_FIXED.

https://stackoverflow.com/questions/6554825/how-do-i-load-and-execute-an-elf-binar

Manual load and execution (2)

4. Use the program headers to find the DYNAMIC vector, which will in turn give you the address of the relocation vectors
5. Apply the relocations
   Assuming your binary was a *static-linked PIE* binary, they should consist entirely of *RELATIVE* relocations (just adding the base load address), meaning you don't have to perform any symbol lookups or anything fancy.

`https://stackoverflow.com/questions/6554825/how-do-i-load-and-execute-an-elf-binar`

Manual load and execution (3)

6. Construct an ELF program entry stack consisting of the following sequence of system-word-sized values in an array on the stack:

   ```
   ARGC ARGV[0] ARGV[1] ... ARGV[ARGC-1] 0 \

   ENVIRON[0] ENVIRON[1] ... ENVIRON[N] 0 0
   ```

7. (This step requires ASM!)
   Point the stack pointer at the beginning of this array and jump to the loaded program's entry point address (which can be found in the program headers).

https://stackoverflow.com/questions/6554825/how-do-i-load-and-execute-an-elf-bina

ELF program header (1)

- The program header table tells the system how to create a process image
- it is found at file offset `e_phoff` and consists of `e_phnum` entries each with size `e_phentsize`
- The layout is slightly different in 32-bit ELF vs 64-bit ELF, because the `p_flags` are in a different structure location for alignment reasons.

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF program header (2)

- The Program Header Table contains information for the kernel on how to start the program.
- the LOAD directives specifies a loadable segment parts of the ELF file get mapped into memory
- The INTERP directive specifies an ELF interpreter normally /lib/ld-linux.so.2
- The DYNAMIC entry points to the .dynamic section contains information used by the ELF interpreter to setup the binary

https://www.ics.uci.edu/~aburtsev/143A/hw/hw2/hw2-elf.html

ELF program header example

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0xa0200 0xa0200 R E 0x1000
  LOAD           0x0a0200 0x080e9200 0x080e9200 0x04b44 0x09728 RW  0x1000
  DYNAMIC        0x0a0214 0x080e9214 0x080e9214 0x000d8 0x000d8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00020 0x00020 R   0x4
  GNU_EH_FRAME   0x0a0138 0x080e8138 0x080e8138 0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

https://greek0.net/elf.html

ELF program header fields (1)

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
// 52 bytes for 32-bit machines
```

```
typedef struct {
        Elf64_Word      p_type;
        Elf64_Word      p_flags;
        Elf64_Off       p_offset;
        Elf64_Addr      p_vaddr;
        Elf64_Addr      p_paddr;
        Elf64_Xword     p_filesz;
        Elf64_Xword     p_memsz;
        Elf64_Xword     p_align;
} Elf64_Phdr;
// 64 bytes for 64-bit machines
```

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF program header fields (2)

- p_type : the kind of segment
  this array element describes or how to interpret
  the array element's information.
- p_offset : the offset from the beginning of the file
  at which the first byte of the segment resides

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF program header fields (3)

- p_vaddr : the virtual address
  at which the first byte of the segment resides in memory.
- p_paddr : the segment's physical address for systems
  in which physical addressing is relevant
  - the system ignores physical addressing
    for application programs,
  - this member has unspecified contents
    for executable files and shared objects

`https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

ELF program header fields (4)

- p_filesz : the number of bytes
  in the file image of the segment, which can be zero.
- p_memsz : the number of bytes
  in the memory image of the segment, which can be zero.
- p_flags : flags relevant to the segment.

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF program header fields (5)

- p_align : loadable process segments
  must have congruent values
  for `p_vaddr` and `p_offset`, modulo the page size.
    - this member gives the value
      to which the segments are aligned
      in memory and in the file
    - values 0 and 1 mean no alignment is required.
    - otherwise, `p_align` should be a positive,
      integral power of 2,
    - `p_vaddr` should equal `p_offset`, modulo `p_align`

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF program header field p_type (1)

PT_NULL	0	unused
PT_LOAD	1	a loadable segment
PT_DYNAMIC	2	dynamic linking information
PT_INTERP	3	an interpreter path name
PT_NOTE	4	auxiliary information
PT_SHLIB	5	unspecified semantics
PT_PHDR	6	the program header table
PT_LOSUNW	0x6ffffffa	sun microsystems
PT_SUNWBSS	0x6ffffffb	sun microsystems
PT_SUNWSTACK	0x6ffffffa	sun microsystems
PT_HISUNW	0x6fffffff	sun microsystems
PT_LOPROC	0x70000000	a processor specific semantics
PT_HIPROC	0x7fffffff	a processor specific semantics

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

p_type = PT_LOAD segment entry

- specifies a loadable segment, described by
  - p_filesz (the segment's file size) and
  - p_memsz (the segment's memory size)
- The bytes from the file are mapped
  to the beginning of the memory segment
  - case 1) p_memsz > p_filesz,
    the extra bytes are defined to hold the value 0 and
    to follow the segment's initialized area
  - case 2) p_memsz < p_filesz : not possible
- loadable segment entries in the program header table
  appear in ascending order, sorted on the p_vaddr member.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html

p_type = PT_DYNAMIC segment entry

- specifies dynamic linking information

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html

p_type = PT_INTERNP segment entry

- specifies the location and size
  of a null-terminated path name to invoke as an interpreter
- this segment type is mandatory for dynamic executable files and
  can occur in shared objects.
  but cannot occur more than once in a file.
- this type, if present,
  it must precede any loadable segment entry.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html

Sections and segments (1)

- section: tell the linker if a section is either:
  - *raw data* to be loaded into memory,
    e.g. `.data`, `.text`, etc, or
  - formatted *meta data* about other sections,
    that will be used by the linker, but disappear at runtime
    e.g. `.symtab`, `.srttab`, `.rela.text`
- segment: tells the operating system:
  - where should a segment be loaded into *virtual memory*
  - what *permissions* the segments have (read, write, execute).

https://cirosantilli.com/elf-hello-world
https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s

Sections and segments (2)

- ELF files are composed of sections and segments
- sections gather all needed information
  to link a given object file
  and build an executable,
- while Program Headers split the executable
  into segments with different attributes,
  which will eventually be loaded into memory.

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Sections and segments (3)

- segments can be viewed as
  a tool to help the linux loader,
  as they group sections by attributes into single segments
  for more efficient loading process of the executable,
  instead of loading each individual section into memory.

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Sections and segments (4)

- segments' offsets and virtual addresses must be congruent modulo the page size
- their `p_align` field must be a *multiple* of the system page size
- The reason for this alignment is to prevent the mapping of two different segments within a single memory page.

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Sections and segments (5)

- this is due to the fact that different segments usually have different access attributes,
- these cannot be enforced if two segments are mapped within the same memory page.
- therefore, the default segment alignment for `PT_LOAD` segments is usually a system page size
- The value of this alignment will vary in different architecture

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Memory-mapped I/O (1)

- On modern operating systems, it is possible to `mmap` a file to a region of memory then, the file can be accessed just like an array
- This is more efficient than `read` or `write`, as only the regions of the file that a program actually accesses are loaded.

https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html

Memory-mapped I/O (2)

- accesses to not-yet-loaded parts of the mmapped region are handled in the same way as *swapped out pages*.
- since mmapped pages can be *stored back* to their file when physical memory is low, it is possible to `mmap` files orders of magnitude larger than both the physical memory and swap space

https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html

Memory-mapped I/O (3)

- The only limit is address space.
- the theoretical limit is 4GB on a 32-bit machine -
- the actual limit will be smaller
  since some areas will be reserved for other purposes.
- If the LFS (Large File Storage) interface is used
  - the file size on 32-bit systems is not limited to 2GB
  - offsets are signed which reduces the addressable area of 4GB by half
  - the full 64-bit are available.

https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html

Memory-mapped I/O (4)

- `mmap` is a POSIX-compliant Unix system call that maps files or devices into memory.
  - a method of memory-mapped file I/O
  - implements demand paging
- file contents are not read from disk directly and initially do not use physical RAM at all
- the actual reads from disk are performed in a lazy manner, after a specific location is accessed.

https://en.wikipedia.org/wiki/Mmap

Memory-mapped I/O (5)

- after the memory is no longer needed,
  it is important to `munmap` the pointers to it.
- protection information can be managed using `mprotect`
- special treatment can be enforced using `madvise`

https://en.wikipedia.org/wiki/Mmap

Memory-mapped I/O (6)

- demand paging is a method of virtual memory management
  (as opposed to anticipatory paging)
- the os copies a disk page into physical memory
  only if an attempt is made to access it
  and that page is not already in memory (page fault)

https://en.wikipedia.org/wiki/Demand_paging

Memory-mapped I/O (7)

- it follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages are located in physical memory.
- this is an example of a lazy loading technique.

https://en.wikipedia.org/wiki/Demand_paging

Memory-mapped I/O (8)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- `mmap()` creates a new mapping
  in the virtual address space of the calling process
- the starting address for the new mapping is specified in `addr`
- the `length` argument specifies
  the length of the mapping
  (which must be greater than 0).

https://man7.org/linux/man-pages/man2/mmap.2.html

Memory-mapped I/O (9)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- If addr is NULL, then the kernel chooses
  the (page-aligned) address
  at which to create the mapping;
  this is the most portable method of creating a new mapping.
- If addr is not NULL, then the kernel takes it
  as a *hint* about where to place the mapping;

https://man7.org/linux/man-pages/man2/mmap.2.html

Memory-mapped I/O (10)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- on Linux, the kernel will pick a nearby page boundary but always above or equal to the value specified by `/proc/sys/vm/mmap_min_addr` and attempt to create the mapping there.
- If another mapping already exists there, the kernel picks a new address that may or may not depend on the *hint*
- The address of the new mapping is returned as the result of the call.

https://man7.org/linux/man-pages/man2/mmap.2.html

Memory-mapped I/O (11)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- The contents of a file mapping
  (as opposed to an anonymous mapping),
  are initialized using `length` bytes
  starting at `offset` offset in the file (or other object)
  referred to by the file descriptor `fd`

- offset must be a multiple of the page size
  as returned by `sysconf(_SC_PAGE_SIZE)`.

https://man7.org/linux/man-pages/man2/mmap.2.html

Memory-mapped I/O (12)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- After the `mmap()` call has returned,
  the file descriptor, `fd`, can be closed immediately
  without invalidating the mapping.

https://man7.org/linux/man-pages/man2/mmap.2.html

Memory-mapped I/O (13)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- The `prot` argument describes
  the desired memory protection of the mapping
  and must not conflict with the open mode of the file
- It is either `PROT_NONE` or the bitwise `OR` of
  one or more of the following flags:
  - `PROT_EXEC` Pages may be executed.
  - `PROT_READ` Pages may be read.
  - `PROT_WRITE` Pages may be written.
  - `PROT_NONE` Pages may not be accessed.

https://man7.org/linux/man-pages/man2/mmap.2.html