The ELF file format

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
X/GitHub/...: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

Outline

Executable and Linkable Format

Very flexible file format, can be used for

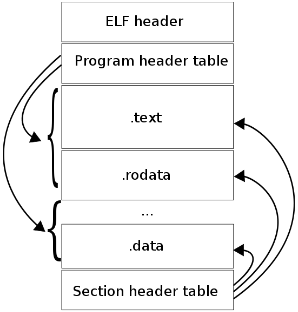
- executables
- shared objects, AKA dynamic libraries
- object files, AKA relocatable files
- core dumps

References:

- `ELF(5)`, `core(5)` and `gcore(1)`
- `en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- `https://refspecs.linuxfoundation.org/elf/elf.pdf`
- `https://uclibc.org/docs/elf-64-gen.pdf`

A nice video on ELF is “In-depth: ELF” `https://youtu.be/nC1U1LJQL8o`

Two views: segments and sections



`https://commons.wikimedia.org/wiki/File:Elf-layout--en.svg`

- ELF header at the beginning is a “road map”
- Section header table, if present, holds linking information: instructions, data, symbol table, relocation information, ...

  Beware: sections *can* be present without their header
- Program header table, if present, describes segments; that is, tells how to build a process image for execution

ELF (file) header

```
typedef struct
{
        unsigned char   e_ident[16];
        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;
```

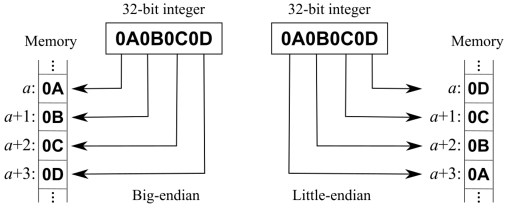
```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  02 00 3e 00 01 00 00 00  50 04 40 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  00 1a 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1f 00 1c 00
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 00 00
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 00 00
```

`https://blog.k3170makan.com/2018/09/introduction-to-elf-format-elf-header.html`

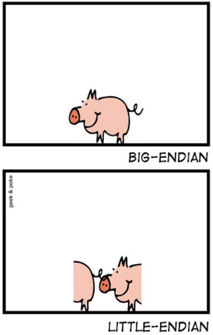
All structure definitions can be found in `/usr/include/elf.h`

Endianness

`e_ident[EI_DATA]` describes the endianness



source: Wikipedia



source: Simply Explained

Intel CPUs use...

little endian

Demos/Exercises

- Change the OS ABI in `hello-world-ok`; does the program still run?
- What's wrong with `hello-world-maybe-broken`?
  - Hint: try comparing the bytes inside the two files (with a hex-editor or `vbindiff`)

Common names of main sections

`.text`	Code
`.data/.rodata/.bss`	Data / read-only data / uninitialized data
`.init/.fini`	Initialization/Termination code
`.init_array/.fini_array`	Pointers to initialization/termination code
`.ctors/.dtors`	Old version of the previous
`.interp`	Interpreter, AKA dynamic-linker
`.dynamic/.got*/.plt*`	"stuff" for dynamic linking
`.debug*`	Debugging information
`*.gnu.*/.gcc*/`	GNU/Linux extensions
`.eh_frame`	Exception handling unwinding information
`.rel*`	Relocations
`*sym*`	Symbols
`*str*`	String tables

Main segment types

- **PT_LOAD** something to "load", typically to `mmap`
- **PT_PHDR** the program header itself, when available in the process memory
- **PT_INTERP** `.interp` section
- **PT_DYNAMIC** `.dynamic` section
- **PT_GNU_EH_FRAME** `.eh_frame_hdr` section, used to locate the `.eh_frame` section
- **PT_GNU_STACK** empty segment, whose flag specify whether the stack should be made executable

Fun with ELF

Some people created very small ELFs, by *abusing* the format. E.g.,

- overlapping different things
- putting *code* inside "holes" in the headers
- omitting trailing zeros
- ...

there are even challenges where the goal is to build the smallest ELF

To know more:

- Write-ups for "PlaidCTF 2020 - golf.so"
  `https://ctftime.org/task/11305`
- "A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux (or, "Size Is Everything")"
  `www.muppetlabs.com/~breadbox/software/tiny/teensy.html`
- "Adventures in Binary Golf - netspooky"
  `https://www.youtube.com/watch?v=VLmrsfSE-tA`

Outline

Program execution

- What happens when (from, for instance, `bash`) we run `./hello-world`?
- Which syscalls are involved when a file gets executed?
  - the shell use `fork(2)` and `execve(2)`, typically through `exec(3)`
- `execve`
  - creates a new (virtual) address space
    - maps the `PT_LOAD` segments
    - creates some runtime segments
    - stack/data regions might be *executable*, depending on segment `GNU_STACK` and kernel version. See: `https://stackoverflow.com/a/64837581`
  - copies filename, command line arguments, and environment into it
  - if the file is an x86/x64 ELF (i.e. corresponding to HW architecture) it can be run
    ...otherwise? → `hello-world.arm`

`https://ownyourbits.com/2018/05/23/the-real-power-of-linux-executables/`

Binary formats

- Linux supports a bunch of binary formats
- Each format is run by a specific handler
  - some handlers come with the standard kernel (ELF, a.out, scripts, ...)
  - others can be added through loadable modules
- Whenever a file is to be executed through `execve`, its 128 first bytes are read and passed on to every handler
  - that can accept or ignore it, depending on some magic value; e.g.
    `0x7F 'E' 'L' 'F'` → ELF
    `#!` → a script
- One, user extensible, handler is binfmt_misc
  - offers a `/proc` interface to the system administrator
    - `sudo sh -c 'echo 1 > /proc/sys/fs/binfmt_misc/status'`
      to enable, 0 to disable
  - specifies what userland interpreter should run for specific file types

Virtual FS: binfmt_misc

- Interpreters specified inside `/proc/sys/fs/binfmt_misc`
  - E.g. `qemu-arm` allows us to run ARM executables on Intel machines (installed by `qemu-user-binfmt`)
- These handlers can be enabled/disabled/removed by writing 1/0/-1 to the corresponding files; e.g. to disables ARM emulation:
  `echo 0 > /proc/sys/fs/binfmt_misc/qemu-arm`
- New entries can be added by writing to `/proc/sys/fs/binfmt_misc/register`; see `https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html`

WSL

On WSL the package `qemu-user-binfmt` does not seem to register the `qemu-arm` interpreter, so arm binaries cannot be run automagically

Outline

Dynamic link libraries

Why?

- Smaller programs, that (automatically) use the most recent libraries
- Code pages can be (ro-)shared among different processes

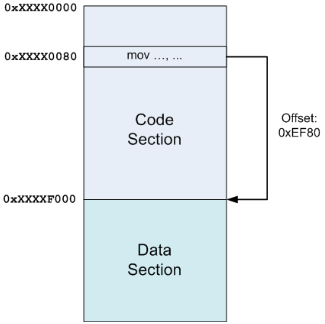
However,

- Different processes may (need to) map libraries at different addresses
- To share pages, we can't relocate by using runtime patching, so:
  - Position Independent Code
    - instead of absolute addresses, EIP/RIP relative addressing
    - in the binary the program base is 0 (and can be mapped everywhere)
  - (Relocatable) external references use indirections through data
    - each external variable/function is accessed *indirectly*
- Libraries "must" be PIC; however, executables ~~are~~were typically position-dependent (more efficient, especially on 32 bits)

How?

Key insights (1/5)

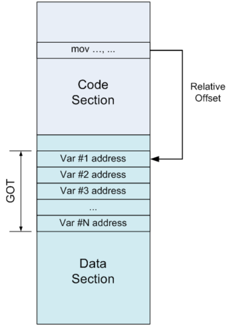
1. offsets between `text` and `data` sections statically known
2. we can use IP-relative offsets to access (statically linked) data
   - requires thunking on 32 bits; x64 can use RIP-relative offsets



Taken from PIC in shared libraries

Key insights (2/5)

3. we use an indirection, through the GOT – Global Offset Table, for dynamically linking external references
   - GOT resides in the `data` section and
     - the (static) linker generates
     - (dynamic) relocation entries for it
   - one relocation entry for each variable v, regardless the number of times v is accessed



Taken from
PIC in shared libraries

Key insights (3/5)

Functions *could* be treated in the same way, *however*

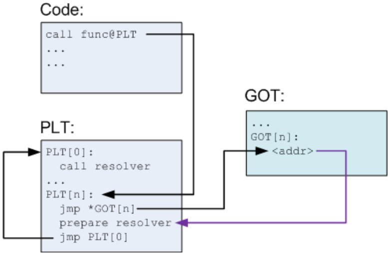
- a program may import *a lot* of functions, and most/some of them could be rarely called
- applying all relocations at startup, slows it down

Idea: lazy binding = fixing relocations on-the-fly when needed

- that is, the first time a function is called
- this is the default for `ld` and *was* the default for `gcc`
  - gcc option: `-z lazy`
- on Ubuntu, `gcc` now tells `ld` to direct `ld.so` to resolve all symbols when the program is started
  - gcc option: `-z now`
  - this is more secure, as we'll discuss

Key insights (4/5)

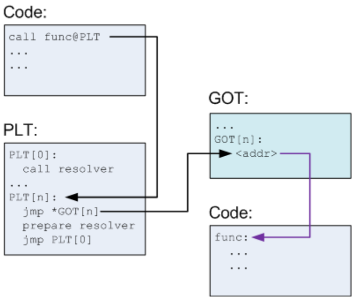
4. calls to dynamic symbols = calls into the Procedure Linkage Table
   - PLT = array of stubs that call the "real" functions using GOT
   - only functions can be lazily bound, variables are always eagerly bound
     - functions, whose address has been taken, are eagerly bound too
5. first entries of PLT/GOT are "special"; PLT[0]/GOT[0] is the resolver



Taken from PIC in shared libraries

Key insights (5/5)

When the symbol has been resolved:



For more details see PIC in shared libraries and [DFCS+15]

Variations

With early-binding:

- depending on gcc version/options, PLT entries could be shortened to 8 bytes, instead of 16 (these are found in `.plt.got` instead of `.plt`):

```
Disassembly of section .plt: [...]
0000000000001040 <printf@plt>:
    1040:   ff 25 8a 2f 00 00       jmp    QWORD PTR [rip+0x2f8a]
    1046:   68 01 00 00 00          push   0x1
    104b:   e9 d0 ff ff ff          jmp    1020 <.plt>

Disassembly of section .plt.got:
0000000000001050 <__cxa_finalize@plt>:
    1050:   ff 25 a2 2f 00 00       jmp    QWORD PTR [rip+0x2fa2]
    1056:   66 90                   xchg   ax,ax
```

- gcc allows to avoid the PLT, option `-fno-plt`, by generating:
  `call QWORD PTR [`*GOT-func*`]` instead of: `call func@PLT`
  - simpler and more efficient

Security considerations GOT/PLT

GOT is an interesting data-structure, which *might* be writable

Food for thought:

- GOT overwrite = calling `system` when you want to call `printf` ☺
- leaking the address of, say `puts`, may help to find the address of `system`

RELRO is a memory corruption mitigation

Related `ld` options:

- `-z norelro`: don't create `PT_GNU_RELRO`
- `-z relro`: create `PT_GNU_RELRO` segment, which will be made read-only after relocation

Full relro protection in gcc: `-z relro -z now`

Building PIC/PIE

GCC options

- `-fpic` generate PIC, which accesses external symbols through a GOT
- `-fpie` similar to `-fpic`, but generated PIC can be only linked into executables; typically used with `-pie`

Be consistent (for predictable results)

`-f...` are for the compiler, `-pie`/`-shared`/`-static` for the linker

- `-static` static linking; shared libraries are ignored
- `-shared -fpic` produce a shared object
- `-pie -fpie` produce an *"executable" shared object* → better ASLR
- *(only in recent gcc)* `-static-pie`

  https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

How to build dynamic libraries

ELF shared objects, created by using `-shared`

- `gcc -shared -fpic -o libfuncs.so my-lib*.c`
- `gcc main.c -L. -lfuncs`
- `LD_LIBRARY_PATH=.  ./a.out`
  - If `LD_LIBRARY_PATH` is defined, it is searched before looking in standard library directories
    - unless the executable is set-UID/GID (more details in the man)
  - a production application should never rely on `LD_LIBRARY_PATH`; shared libraries should be either installed in
    - standard directories, or
    - directories specified by `DT_RUNPATH`/`DT_RPATH` inside the ELF (by using `ld` option `-rpath`). Inside `R(UN)PATH` the name `$ORIGIN` means: "the directory containing the application" — see `ld.so(8)` for more

Outline

Program execution

Dynamically linked executables contain special segments:

`PT_INTERP` the "interpreter" (=the dynamic loader)

`PT_DYNAMIC` linking data

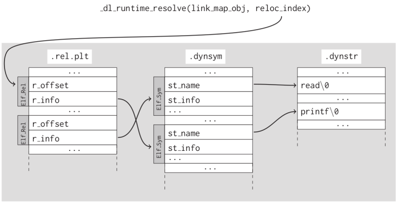
When a program *p* is run

- the kernel creates the address space and maps *p*'s `PT_LOAD` segments
- if there is no `PT_INTERP`, the execution continues in *p*'s entry point
- else, the "interpreter" is mapped along *p* and run

Dynamic tables

`PT_DYNAMIC` contains tags/values; listed by `readelf --dynamic`

- `DT_NEEDED` specifies a needed library
- `DT_STRTAB` points to the dynamic string table
- `DT_SYMTAB` points to the dynamic symbol table
- `DT_PLTREL` specifies whether PLT uses REL or RELA relocations
- `DT_REL[A]` points to the relocations, whose size is given by `DT_REL[A]SZ`



From [DFCS+15]

Dynamic loader

The dynamic linker is `ld.so(8)`, actually

- `ld.so` handles *a.out binaries*
- `ld-linux.so*` handles *ELF*
  - `/lib/ld-linux.so.1` for libc5
  - `/lib/ld-linux.so.2` for glibc2, which has been used for years

  same behavior, and same support files; for details: `ldconfig(8)`

It:

- Finishes mapping needed libraries and then jumps to *p*'s entry-point
- It's a lighter process w.r.t. static linking and uses different sections

In gdb (+GEF) you can observe this process by:

- `set stop-on-solib-events 1`
- `r`
- `info shared` / `vmmap` / `xfiles`

Debugging the dynamic loading

Environment variable LD_DEBUG (ignored in secure-execution mode) can enable the output of debugging information about operation of the dynamic linker.

To get you started:

```
LD_DEBUG=libs /bin/echo BASC
```

For more information: `ld.so(8)`

Symbol resolution

Shared libraries were designed so that the default semantics for symbol resolution exactly mirrored those of static ones

- that is, a previous definition of a global symbol, e.g. in the main program, overrides definitions in following libraries
  - If defined multiple times, bound to the first definition found by scanning libraries in the left-to-right order in which they were listed
- this makes transition from static to shared relatively easy; *however*
- with default semantics, a shared lib is *not* a self-contained subsystem
- to guarantee that an invocation of `foo` in a shared library call its own version of `foo`, we can
  - override default binding with, for instance, `ld -Bsymbolic ...`
  - change default visibility with `-fvisibility=hidden` and/or use `__attribute__((visibility ("default")))`
  - ...

More details in *How To Write Shared Libraries* [Dre11], by Ulrich Drepper, which contains all details and also covers *symbol versioning*

Outline

Library interposition

Linux linkers support *library interposition*:

- intercepting calls to library functions, and executing your own code
- basic idea: calls to a *target function* are replaced with calls to a wrapper function, with the same signature

Three kinds:

- compile-time: using macros of C preprocessor... boring ☺
- link-time: using `--wrap` flag in `ld` (typically through `-Wl,--wrap,`*func-name* from `gcc`)
  - any undefined reference to *symbol* will be resolved to `__wrap_`*symbol*. Any undefined reference to `__real_`*symbol* will be resolved to *symbol*
- run-time, using the linker API (→ next slide)

Dynamic Linker API

```
void *dlopen(const char *filename, int flags);
```

explicitly loads the dynamic shared object (shared library) `filename`, and returns an opaque *handle* for the loaded object

```
void *dlsym(void *handle, const char *symbol);
```

takes a *handle* and a symbol name, and returns the address where `symbol` is loaded into memory

There are two special pseudo-handles:

- `RTLD_DEFAULT`: find the first occurrence of the desired symbol using the default shared object search order
- `RTLD_NEXT`: find the next occurrence of the desired symbol in the search order after the current object. This allows to provide a wrapper around a function in another shared object

LD_PRELOAD

Environment variable `LD_PRELOAD` specifies shared objects to be loaded *before* all others; can be used to override functions

- Indeed, this technique was used by some recent malware; e.g., `https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearly-impossible-to-detect-linux-threat`
- To debug, use:

  strace `-E` to set environment variables for the traced command

  gdb `set environment LD_PRELOAD=...`
- `LD_PRELOAD` ignored for set-UID/GID programs (see man for details)

Interposition example

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

int strcmp(const char *s1, const char *s2)
{
        static int (*real_strcmp)(const char *, const char *) = 0;
        if (!real_strcmp)
                real_strcmp = dlsym(RTLD_NEXT, "strcmp");

        int result = real_strcmp(s1, s2);
        fprintf(stderr, "strcmp(%s, %s)=%d\n", s1, s2, result);
        /* return result; */
        return 0;
}
```

→ `interposition`

Exercise: antidebug1

1. `./antidebug1`
2. `strace -o /dev/null ./antidebug1`

Isn't that suspicious? Let's analyze the behavior:

- let's try: `strace ./antidebug1`
  - the interesting part is almost at the end of the output
- can you bypass the check?
  - On my Ubuntu the actual prototype for current implementation is:
    `long int ptrace(enum __ptrace_request __request, ...)`
    (see `/usr/include/x86_64-linux-gnu/bits/ptrace-shared.h`)

You can also take a look at `antidebug{2,3}`
Beware: you can't solve all of them
(with the tools we have seen so far. . . stay tuned ☺)

Outline

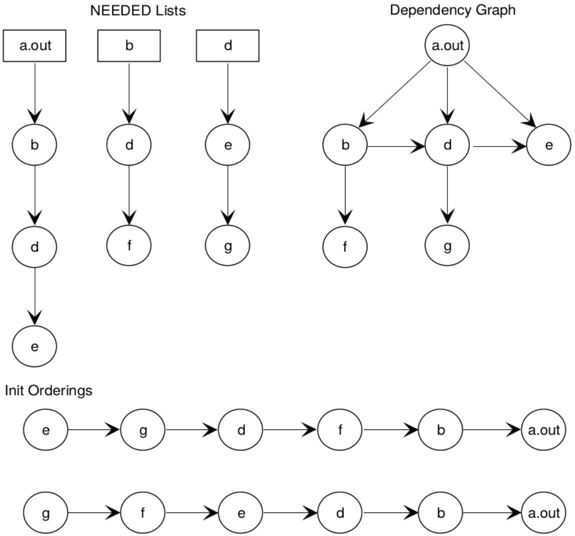
Initialization and termination order

From ELF standard:

- After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code
  - All s.o. initializations happen before the executable gains control
- Before the initialization code for any object A is called, the initialization code for any other objects that object A depends on are called. For these purposes, an object A depends on another object B, if B appears in A's list of needed objects (recorded in the `DT_NEEDED` entries of the dynamic structure)
  - The order of initialization for circular dependencies is undefined

Example. . .

Construction order example

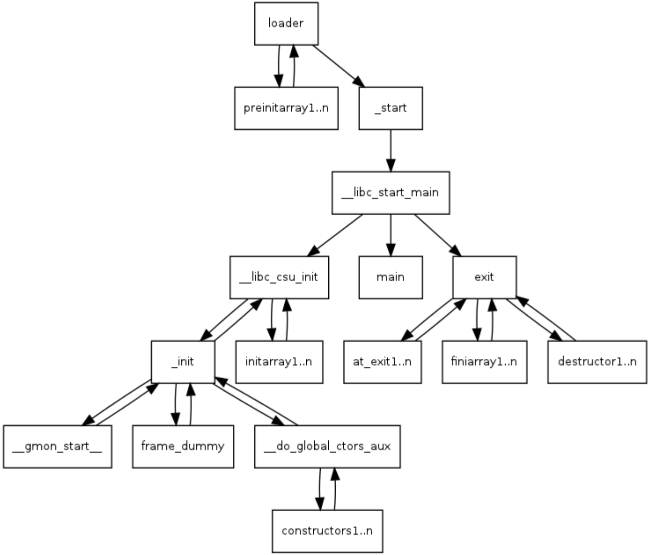


Initialization and termination sections

- Two section types:
  - array of pointers: `.preinit_array`, `.init_array` and `.fini_array`
    - the link-editor encodes these information inside `DYNAMIC` segment
    - Gcc used also a similar `.ctors`
  - obsolete, a single code block: `.init`

  built by concatenating like sections from relocatable objects
- The runtime linker (or a startup mechanism) executes:
  1. functions whose addresses are contained in the `.preinit_array`
     - for executables only
  2. the `.init` section, as an individual function, `_init`
  3. functions whose addresses are contained in the `.init_array`
- Analogously, for termination
  - `.fini_array`, `.fini` (and `.dtors`)

Going down to the rabbit hole



Constructor/destructor example

```c
#include <stdio.h>

__attribute__((constructor))
void foo() {
        printf("Hello!\n");
}

__attribute__((destructor))
void bar() {
        printf("Bye\n");
}

int main()
{
        printf("... in main ...\n");
}
```

More resources

- The ELF file format
  `https://www.gabriel.urdhr.fr/2015/09/28/elf-file-format/`
- ELF loading and dynamic linking
  `https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/`
- Anatomy of an ELF core file
  `https://www.gabriel.urdhr.fr/2015/05/29/core-file/`
- Preeny, a collection of `LD_PRELOAD`ed CTF-oriented “tricks”:
  `https://github.com/zardus/preeny`

References

[DFCS+15] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
How the ELF Ruined Christmas.
In *USENIX Security Symposium*, pages 643–658, 2015.

[Dre11] Ulrich Drepper.
How to write shared libraries, 2011.