Binary Analysis - 1

REVERSE ENGINEERING

João Paulo Barraca

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

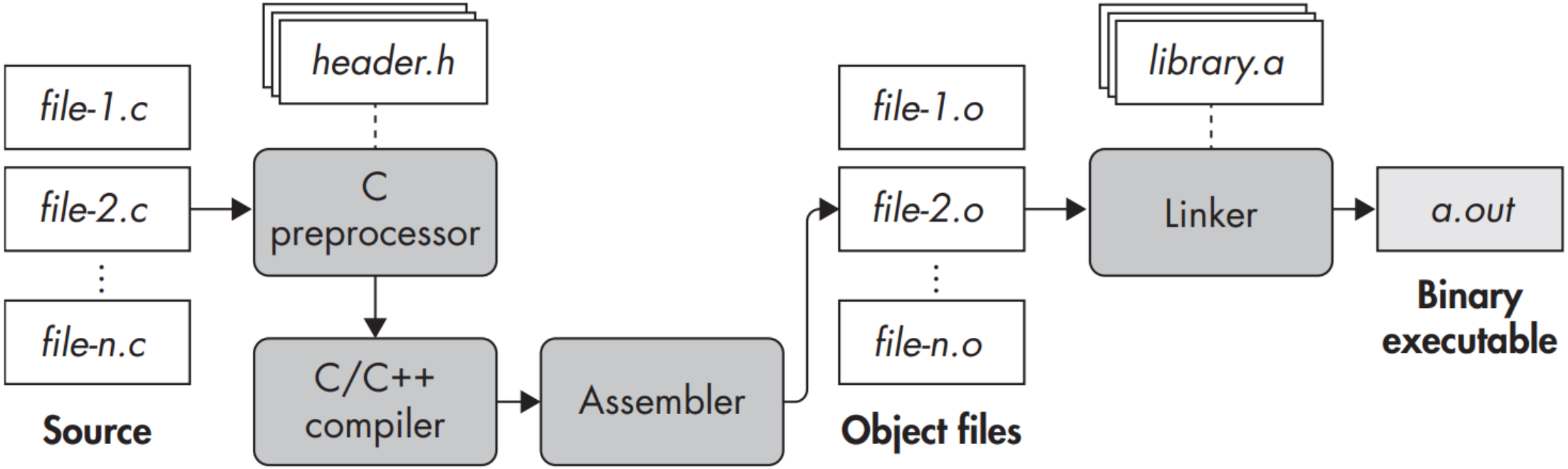
Binary Objects

Binary files

- The result of a compilation process
  - Translating high level code (C/C++, etc…) into native code or bytecode
- Code is encapsulated in a binary format
  - It’s not a raw file with unstructured bytes
- Target system (CPU or VM) will process the resulting code
  - Which may be only part of the file content

Compilation process

The C/C++ use case

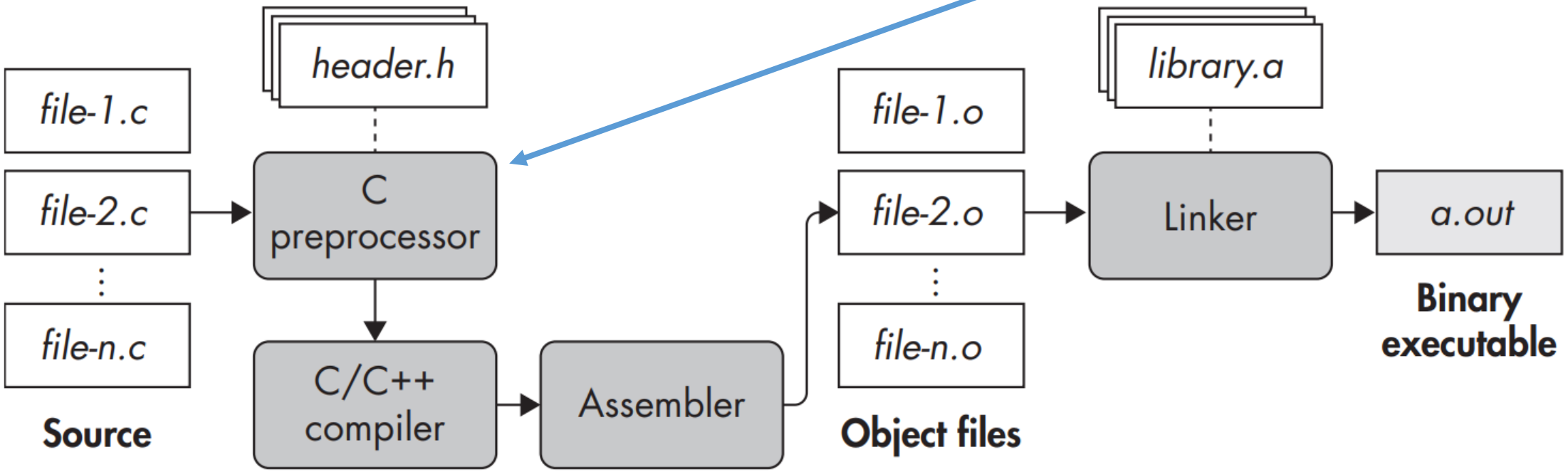


Compilation process

The C/C++ use case

Pre-processor (may be the compiler) processes code, validating its structure and expanding existing macros.

Result is a text blob with content ready to be further processed, and frequently without external dependencies



Hello.c

```c
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char** argv) {
    printf("Hello World\n");

    return 0;
}

```

Hello.c

Pre-compile: gcc –E –o hello.e hello.c
produces >1500 lines

```
1760  extern int rpmatch (const char *__response) __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1))) ;
1761  # 954 "/usr/include/stdlib.h" 3 4
1762  extern int getsubopt (char **__restrict __optionp,
1763            char *const *__restrict __tokens,
1764            char **__restrict __valuep)
1765        __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1, 2, 3))) ;
1766  # 1000 "/usr/include/stdlib.h" 3 4
1767  extern int getloadavg (double __loadavg[], int __nelem)
1768        __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1)));
1769  # 1010 "/usr/include/stdlib.h" 3 4
1770  # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
1771  # 1011 "/usr/include/stdlib.h" 2 3 4
1772  # 1020 "/usr/include/stdlib.h" 3 4
1773
1774  # 3 "hello.c" 2
1775
1776
1777
1778  # 5 "hello.c"
1779  int main(int argc, char** argv) {
1780   printf("Hello World\n");
1781
1782   return 0;
1783  }
```

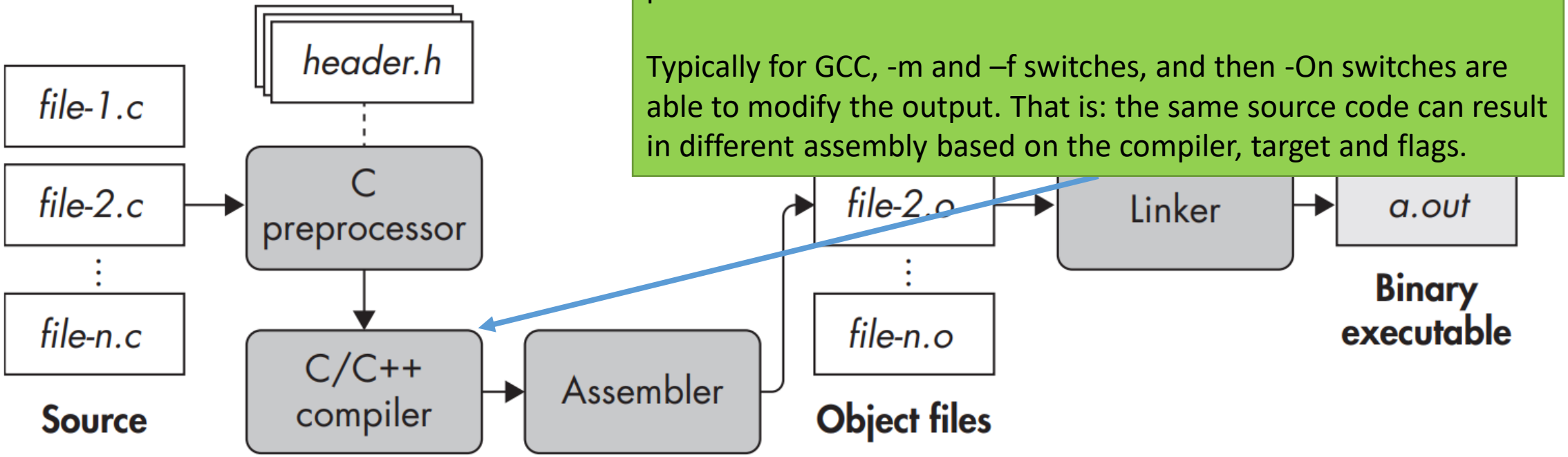
Compilation process

The C/C++ use case

Compiler processes the file and produces assembly code. This may result in assembly for an intermediate processor, and not the final processor.

The processor will create abstract syntax trees (AST) and may tweak or optimize the result according to the options it was provided with.

Typically for GCC, -m and –f switches, and then -On switches are able to modify the output. That is: the same source code can result in different assembly based on the compiler, target and flags.



Hello.c

Compile: gcc -masm intel -S –o hello.s hello.c

File Metadata

Constant variables and symbols

Compiler additional data. In this case Call Frame Information to handle exceptions

Assembly instructions. Notice that symbols are kept as labels

Additional sections to produce:
Entry point
Compiler identification
Instruct linker to mark stack as NX

```
1   .file   "hello.c"
2           .intel_syntax noprefix
3           .text
4           .section        .rodata
5   .LC0:
6           .string "Hello World"
7           .text
8           .globl  main
9           .type   main, @function
10  main:
11  .LFB6:
12          .cfi_startproc
13          push    rbp
14          .cfi_def_cfa_offset 16
15          .cfi_offset 6, -16
16          mov     rbp, rsp
17          .cfi_def_cfa_register 6
18          sub     rsp, 16
19          mov     DWORD PTR -4[rbp], edi
20          mov     QWORD PTR -16[rbp], rsi
21          lea     rdi, .LC0[rip]
22          call    puts@PLT
23          mov     eax, 0
24          leave
25          .cfi_def_cfa 7, 8
26          ret
27          .cfi_endproc
28  .LFE6:
29          .size   main, .-main
30          .ident  "GCC: (Debian 8.3.0-6) 8.3.0"
31          .section        .note.GNU-stack,"",@progbits
```

Compilation process

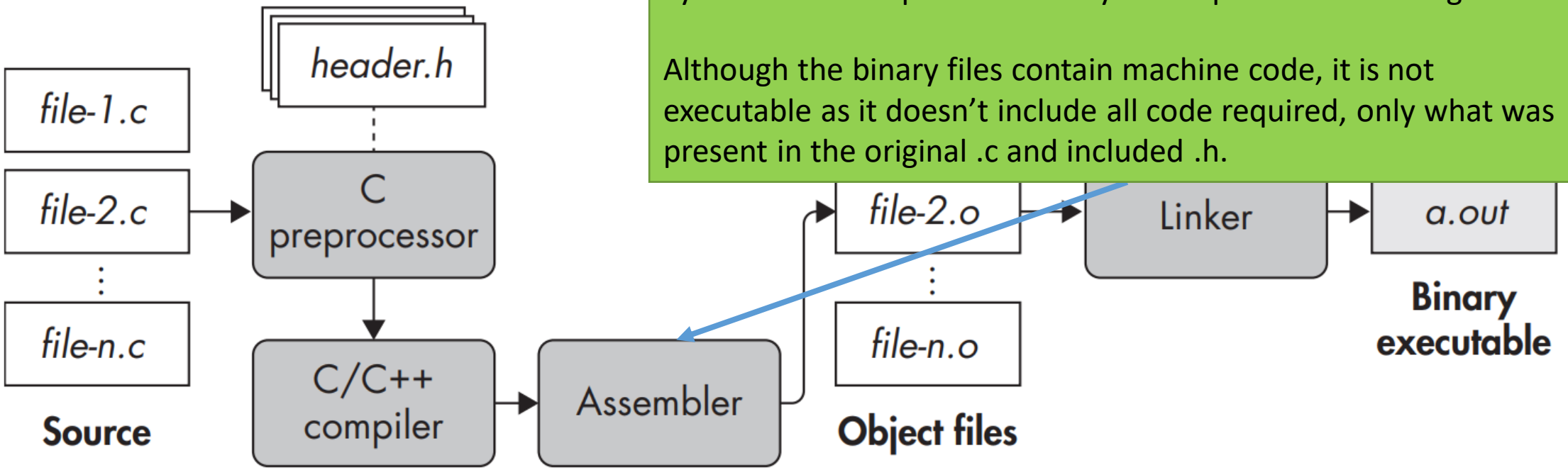
The C/C++ use case

Input containing assembly code is transformed into machine code. Output is a set of object files, or modules with a .o extension.

Code produced may use relative addresses, making it reusable (technically *relocatable)* when integrated into a final binary file.

Symbols are also present as they are required at later stages.

Although the binary files contain machine code, it is not executable as it doesn't include all code required, only what was present in the original .c and included .h.



Hello.c

Compile: gcc -c –o hello.o hello.c

64 bit, Least Significant Byte (Little Endian)

Not stripped = Contains symbols

Assemble the code into machine code

```
$ gcc -c -o hello.o hello.c

$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

$ ./hello.o
-bash: ./hello.o: cannot execute binary file: Exec format error

$ nm hello.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U puts
```

File is an Executable and Linkable Format (ELF)

Cannot be executed

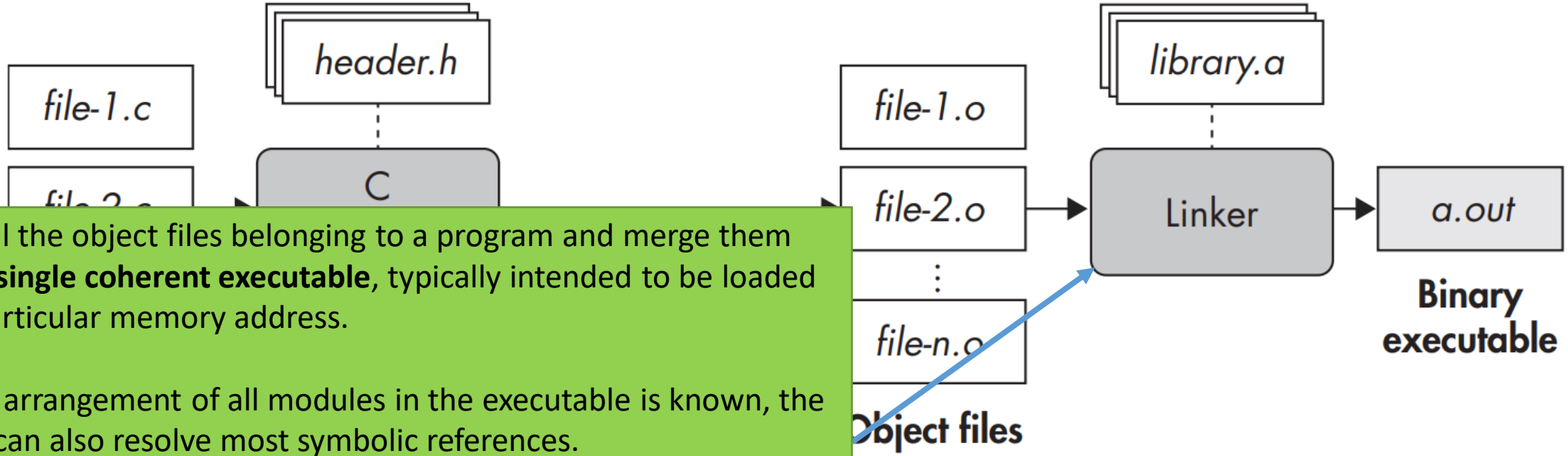
Defines a symbol **`main`** in the Text section

`_GLOBAL_OFFSET_TABLE_` and **`puts`** are not defined.
Code is not present on the object file

SYSV = System V ABI. Identifies the target system (others: Solaris, Tru64, FreeBSD, NetBSD…)

Compilation process

The C/C++ use case



Take all the object files belonging to a program and merge them into a **single coherent executable**, typically intended to be loaded at a particular memory address.

As the arrangement of all modules in the executable is known, the linker can also resolve most symbolic references.

References to libraries may or may not be completely resolved, depending on the type of library. In this case, the library is added as a dependency and the symbol is resolved in real time.

Hello.c

Compile: gcc –o hello hello.c

- 64 bit, Little Endian Architecture
- Position Independent Executable (Can use ALSR)
- Uses shared libraries
- Uses the ld-linux-x86-64.so.2 loader
- sha1 build id
- Not stripped: contains symbol names

```
1  $ gcc -o hello hello.c
2
3  $ file hello
4  hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
5  /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=2d3c19e9d0110eef7554245eb02d70bcc9b60dd2, not stripped
6
7  $ ldd hello
8          linux-vdso.so.1 (0x00007fffdb8a1000)
9          libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3e844a0000)
10         /lib64/ld-linux-x86-64.so.2 (0x00007f3e84685000)
```

Shared libraries required to execute this file. Some code is not on the hello binary and is on the libraries

Hello.c

```
$ nm hello

0000000000004030 B __bss_start
0000000000004030 b completed.7325
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000004020 D __data_start
0000000000004020 W data_start
0000000000001080 t deregister_tm_clones
00000000000010f0 t __do_global_dtors_aux
0000000000003df0 t __do_global_dtors_aux_fini_array_entry
0000000000004028 D __dso_handle
0000000000003df8 d _DYNAMIC
0000000000004030 D _edata
0000000000004038 B _end
00000000000011c4 T _fini
0000000000001130 t frame_dummy
0000000000003de8 t __frame_dummy_init_array_entry
0000000000002154 r __FRAME_END__
0000000000004000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002010 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
0000000000003df0 t __init_array_end
0000000000003de8 t __init_array_start
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
00000000000011c0 T __libc_csu_fini
0000000000001160 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000001135 T main
                 U puts@@GLIBC_2.2.5
00000000000010b0 t register_tm_clones
0000000000001050 T _start
0000000000004030 D __TMC_END__
```

Symbols present in the file

Bb: in the BSS
D: in the initialized data Sec.
Rr: in the Read Only Data Sec.
Tt: in the Text (code) Sec.
U: Undefined
Ww: Weak
- default impl. to be overridden

Some are undefined.
Will be defined by the dynamic linker. Code resides on an external object.

Executable Symbols

Tables

- Symbols are names identifying addresses of a binary
  - Have a type, such as Function, and including Undefined
  - E.g. functions create symbols, especially external functions (puts)

- ELF files have two symbol tables
  - .dynsym: symbols which will be allocated to memory when the program loads.
    - In the example, puts is provided by libc, required for operation, and exists as a dynamic symbol
  - .symtab: contains all symbols, including many used for linking and debugging, but not related to code required for execution.
    - These areas will not be allocated (mapped) to RAM
    - Extremely useful to identify the name of functions/sections when reversing!

Executable Symbols

Stripping

- Only symbols in the .dyntab are required
  - Identify allocated sections
  - Identify symbols that must be resolved in external libraries
  - Used for Dynamic Linking when the program is loaded

- **Stripping** is the process of removing unused symbols and code from a binary
  - Stripped binaries take less space, and are not reversed so easily
    - There is no hints about the purpose of a function from its name

Hello

```
$ readelf --syms hello

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 64 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000000002a8     0 SECTION LOCAL  DEFAULT    1
     2: 00000000000002c4     0 SECTION LOCAL  DEFAULT    2
   ...
    48: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
    49: 0000000000004030     0 NOTYPE  GLOBAL DEFAULT   24 _edata
    50: 00000000000011c4     0 FUNC    GLOBAL HIDDEN    15 _fini
    51: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    52: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    53: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    54: 0000000000004028     0 OBJECT  GLOBAL HIDDEN    24 __dso_handle
    55: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    56: 0000000000001160    93 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    57: 0000000000004038     0 NOTYPE  GLOBAL DEFAULT   25 _end
    58: 0000000000001050    43 FUNC    GLOBAL DEFAULT   14 _start
    59: 0000000000004030     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    60: 0000000000001135    34 FUNC    GLOBAL DEFAULT   14 main
    61: 0000000000004030     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
    62: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    63: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC_2.2
```

Hello.c

Binary is stripped of extra symbols

Only the `.dynsym` table is kept
Required for identifying allocatable areas
Notice as all symbols here are undefined (must be dynamically linked)

```
$ strip hello

$ readelf --syms hello

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
```

What is inside an Object File?

- An Object File contains information required to execute a program (not only code)
  - May not include all implementation, as this can be dynamically loaded

- Information is kept in sections, which are processed differently. Some are:
  - **.rodata:** readonly data, containing strings
  - **.got:** Global Offset Table - maps symbols to memory locations (offsets).
  - **.plt:** Procedure Linkage Table – uses the PLT to transfer execution to the correct location of a symbol, dealing with external symbols and fixing the GOT
  - **.bss: Block Starting Symbol –** contains uninitialized variables
  - **.dynsym:** List of symbols in allocatable memory
  - **… many others:**
    - **To read sections:** `readelf -S hello`
    - **To dump all code:** `objdump -M intel -d hello`

Hello ELF content

RODATA: objdump -sj .rodata

Contains Read Only Data (Strings and other constants)

```
1  $ objdump -sj .rodata hello
2
3  hello:     file format elf64-x86-64
4
5  Contents of section .rodata:
6   2000 01000200 48656c6c 6f20576f 726c6400  ....Hello World.
```

Hello ELF disassembly

Indirection at PLT

The entry point to the program.
Prepares stack
Calls **`main`** function

The main function:
Allocates `0x10` in the stack
Sets arguments to puts
Calls `puts@PLT`
Sets the Return Code to 0
Leave

```
objdump -M intel -d hello

hello:     file format elf64-x86-64

...

0000000000001030 <puts@plt>:
    1030:       ff 25 e2 2f 00 00       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
    1036:       68 00 00 00 00          push   0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <.plt>

...

0000000000001050 <_start>:
    1050:       31 ed                   xor    ebp,ebp
    1052:       49 89 d1                mov    r9,rdx
    1055:       5e                      pop    rsi
    1056:       48 89 e2                mov    rdx,rsp
    1059:       48 83 e4 f0             and    rsp,0xfffffffffffffff0
    105d:       50                      push   rax
    105e:       54                      push   rsp
    105f:       4c 8d 05 5a 01 00 00    lea    r8,[rip+0x15a]        # 11c0 <__libc_csu_fini>
    1066:       48 8d 0d f3 00 00 00    lea    rcx,[rip+0xf3]        # 1160 <__libc_csu_init>
    106d:       48 8d 3d c1 00 00 00    lea    rdi,[rip+0xc1]        # 1135 <main>
    1074:       ff 15 66 2f 00 00       call   QWORD PTR [rip+0x2f66]        # 3fe0 <__libc_start_main@GLIBC_2.2.5>
    107a:       f4                      hlt
    107b:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]

0000000000001135 <main>:
    1135:       55                      push   rbp
    1136:       48 89 e5                mov    rbp,rsp
    1139:       48 83 ec 10             sub    rsp,0x10
    113d:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
    1140:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
    1144:       48 8d 3d b9 0e 00 00    lea    rdi,[rip+0xeb9]        # 2004 <_IO_stdin_used+0x4>
    114b:       e8 e0 fe ff ff          call   1030 <puts@plt>
    1150:       b8 00 00 00 00          mov    eax,0x0
    1155:       c9                      leave
    1156:       c3                      ret
    1157:       66 0f 1f 84 00 00 00    nop    WORD PTR [rax+rax*1+0x0]
    115e:       00 00
```

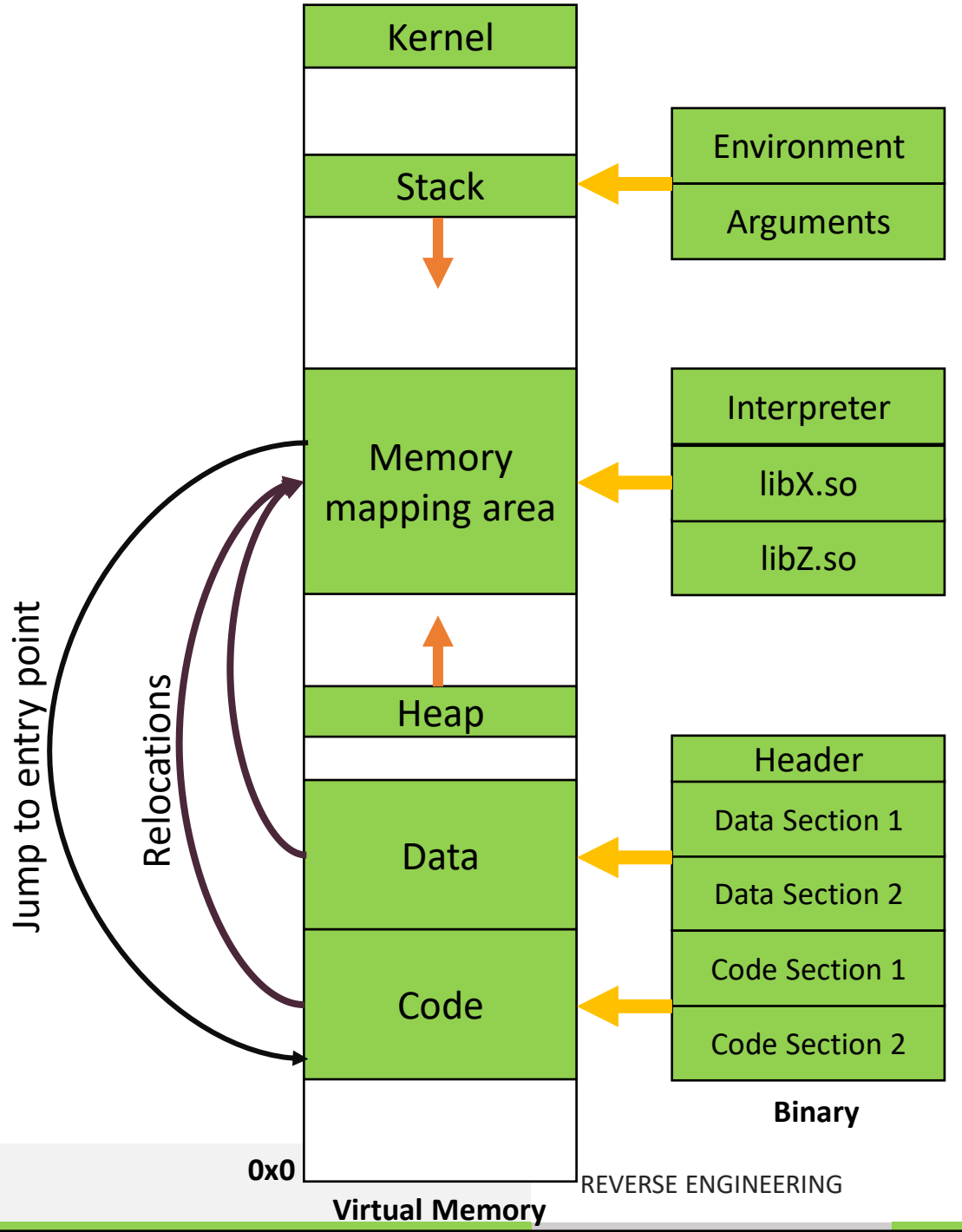
Hello Relocations

readelf --relocs hello

```
1  $ readelf --relocs hello
2
3  ...
4
5  Relocation section '.rela.plt' at offset 0x548 contains 1 entry:
6    Offset          Info           Type           Sym. Value    Sym. Name + Addend
7  000000004018  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

How are objects loaded?

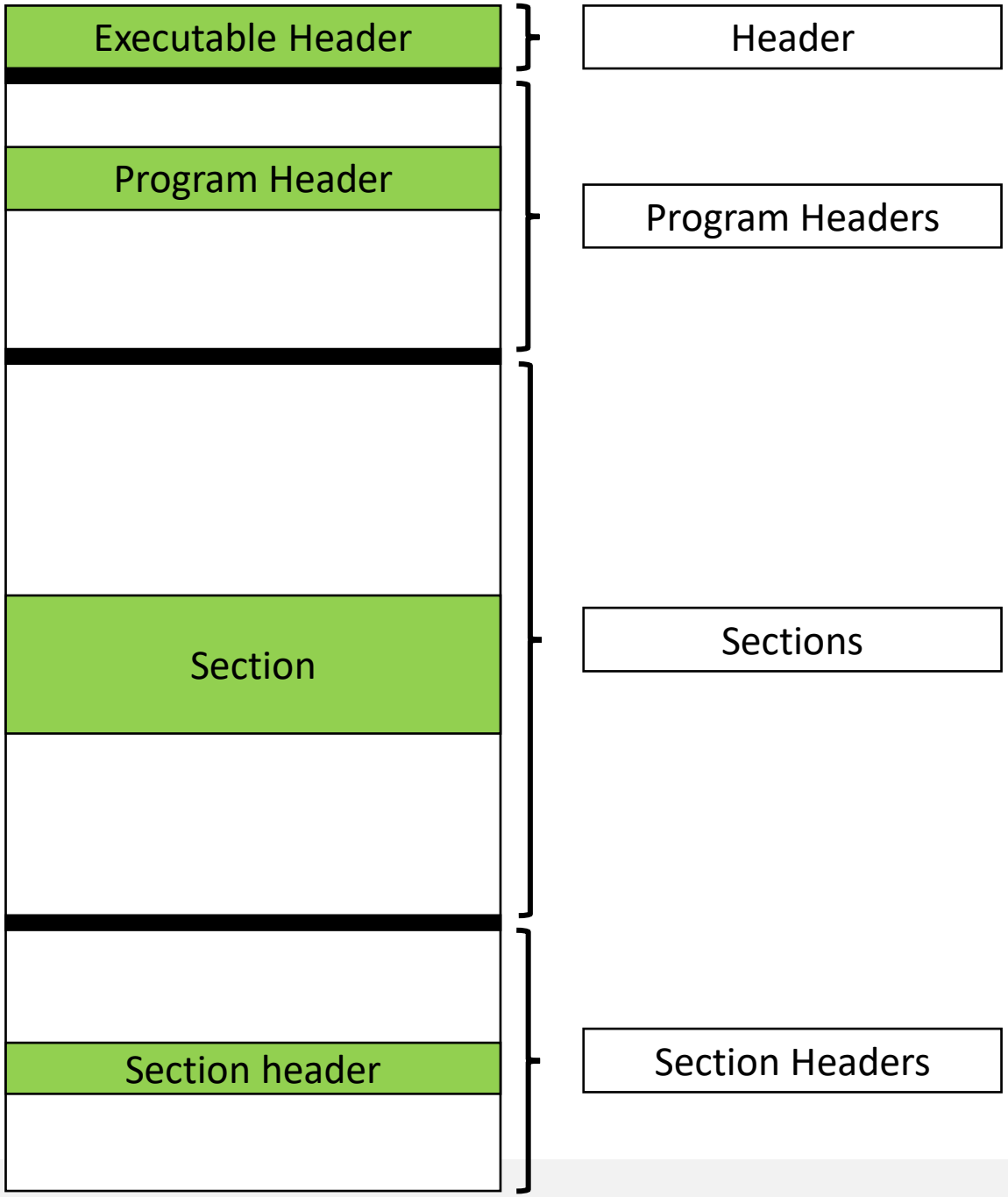
- File is split according to existing sections
  - Each loaded at a different location (with different access attributes)
- Libraries are also mapped in the program address space
  - All code from libraries is present
- Stack grows downwards, heap grows upwards
  - On modern SOs, growth may be limited, not on microcontrollers
- Interpreter is required to setup the binary in memory
  - `ld-Linux.so or ntdll.dll`
    - `readelf -p .interp filename`
  - Will handle relocations, resolving required symbols
  - If lazy-loading is used, relocation is done when the symbol is first used



ELF Files

ELF – Executable and Linkable Format

- Container for executable files, object files, shared libraries, and core dumps
  - And other things out of this context like in Android
- Composed by several headers and sections:
  - Executable Header
  - Several Program Headers (optional)
  - Several Sections, with a header and content


Executable Header
Program Header
Section
Section header
Header
Program Headers
Sections
Section Headers

ELF Headers

Executable Header

- Mandatory header, with basic information about the file
  - Architecture
  - Entry Point
  - Header locations and number
  - Type
  - Type of data

- Follow the structure **`Elf64_Ehdr`**
  - defined in `/usr/include/elf.h`

```
$ readelf -h hello

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:               0x1050
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14688 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

ELF Headers

Section Headers

- Sections are unstructured placeholders of data (frequently code) **targeting the Linker**
  - Some sections are well known and follow a defined structure
  - Some sections can be arbitrary binary blob
  - Some sections may contain content not useful for execution
  - Section order is irrelevant
  - Symbols, relocation information is stored in sections
- Section headers describe the properties of each section
  - Name, type, flags, address when loaded, file offset, size, information…
- ELF files that require no linking, may omit section headers

```
$ readelf -S hello |grep "\["

  [Nr] Name              Type             Address           Offset
  [ 0]                   NULL             0000000000000000  00000000
  [ 1] .interp           PROGBITS         00000000000002a8  000002a8
  [ 2] .note.ABI-tag     NOTE             00000000000002c4  000002c4
  [ 3] .note.gnu.build-i NOTE             00000000000002e4  000002e4
  [ 4] .gnu.hash         GNU_HASH         0000000000000308  00000308
  [ 5] .dynsym           DYNSYM           0000000000000330  00000330
  [ 6] .dynstr           STRTAB           00000000000003d8  000003d8
  [ 7] .gnu.version      VERSYM           000000000000045a  0000045a
  [ 8] .gnu.version_r    VERNEED          0000000000000468  00000468
  [ 9] .rela.dyn         RELA             0000000000000488  00000488
  [10] .rela.plt         RELA             0000000000000548  00000548
  [11] .init             PROGBITS         0000000000001000  00001000
  [12] .plt              PROGBITS         0000000000001020  00001020
  [13] .plt.got          PROGBITS         0000000000001040  00001040
  [14] .text             PROGBITS         0000000000001050  00001050
  [15] .fini             PROGBITS         00000000000011c4  000011c4
  [16] .rodata           PROGBITS         0000000000002000  00002000
  [17] .eh_frame_hdr     PROGBITS         0000000000002010  00002010
  [18] .eh_frame         PROGBITS         0000000000002050  00002050
  [19] .init_array       INIT_ARRAY       0000000000003de8  00002de8
  [20] .fini_array       FINI_ARRAY       0000000000003df0  00002df0
  [21] .dynamic          DYNAMIC          0000000000003df8  00002df8
  [22] .got              PROGBITS         0000000000003fd8  00002fd8
  [23] .got.plt          PROGBITS         0000000000004000  00003000
  [24] .data             PROGBITS         0000000000004020  00003020
  [25] .bss              NOBITS           0000000000004030  00003030
  [26] .comment          PROGBITS         0000000000000000  00003030
  [27] .symtab           SYMTAB           0000000000000000  00003050
  [28] .strtab           STRTAB           0000000000000000  00003650
  [29] .shstrtab         STRTAB           0000000000000000  00003853
```

ELF Sections

.init and .fini

- Contains executable code required before/after the binary entry point is executed
  - Initialization tasks to prepare/clean the memory space

- Some uses:
  - prepare profiling tasks (`__gmon_start__`)
  - Invoke global constructors/destructors (C++)
  - Save program arguments

```
$ objdump -M intel -d -j .init hello

hello:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       48 83 ec 08             sub    rsp,0x8
    1004:       48 8b 05 dd 2f 00 00    mov    rax,QWORD PTR [rip+0x2fdd]        # 3fe8 <__gmon_start__>
    100b:       48 85 c0                test   rax,rax
    100e:       74 02                   je     1012 <_init+0x12>
    1010:       ff d0                   call   rax
    1012:       48 83 c4 08             add    rsp,0x8
    1016:       c3                      ret
```

ELF Sections

.text section

- Contains the main program code
  - The main target of a Reverse Engineering activity
  - Allocated as executable and read-only
  - Contains the user code, and additional code created by the compiler
    - Cleanup/initialization functions, stack guards, etc..
- In this section resides the program entry point
  - When the binary is loaded, execution flow is transferred that address
  - **Related** to the **`main`** function in a C program (but not the main)

```
13    Entry point address:               0x1050
```

ELF Sections

.text section: Entry Point

```
$ objdump -M intel -d -j .text hello

hello:     file format elf64-x86-64


Disassembly of section .text:

0000000000001050 <_start>:
    1050:       31 ed                   xor    ebp,ebp
    1052:       49 89 d1                mov    r9,rdx
    1055:       5e                      pop    rsi
    1056:       48 89 e2                mov    rdx,rsp
    1059:       48 83 e4 f0             and    rsp,0xfffffffffffffff0
    105d:       50                      push   rax
    105e:       54                      push   rsp
    105f:       4c 8d 05 5a 01 00 00    lea    r8,[rip+0x15a]        # 11c0 <__libc_csu_fini>
    1066:       48 8d 0d f3 00 00 00    lea    rcx,[rip+0xf3]        # 1160 <__libc_csu_init>
    106d:       48 8d 3d c1 00 00 00    lea    rdi,[rip+0xc1]        # 1135 <main>
    1074:       ff 15 66 2f 00 00       call   QWORD PTR [rip+0x2f66]        # 3fe0 <__libc_start_main@GLIBC_2.2.5>
    107a:       f4                      hlt
    107b:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]

...
```

The **hello** program entry point address

Loads the address of the main function into **RDI** (first argument) of a function

Calls __libc_start_main@GLIBC_2.2.5 which transfers control to the program **main** function

ELF Sections

.bss, .data, .rodata

- `.rodata`: Read only data
  - Stores constant values
  - Mapped to a page marked as read only
- `.data`: Area with information to initialize variables
  - As the data can be modified, the section is writable
- `.bss`: Unitialized variables
  - Memory is allocated for a variable that may be required, but nothing else is done
  - As there is no data associated, the .bss doesn't take space on the binary. Only instructs the system to reserve memory.

ELF Sections

.plt, .got, .got.plt

- **Procedure Linkage Table** and **Global Offset Table**
  - .**PLT**: Code to relocate symbols
  - .**GOT**: Array with addresses of each symbol requiring relocation
    - .**got** is similar to **.got.plt** but it's writable, while **.got** may be marked as Read Only as a security measure (-z relro)
    - Using a table (GOT) allows patching this table, while keeping libraries in same address, shared to multiple processes

- Sections required for lazy binding (real time relocation)
  - Linker needs to resolve the effective address of a code identified by a symbol (e.g **`puts`**)
  - The code may be on the program, or on an external library, mapped to the virtual memory
  - **.plt** and **.got** ensure the symbol location is found and the code jumps around correctly
  - This is executed as the symbols are required! (LAZY)
  - On Linux, the Env Variable `LD_BIND_NOW` forces linking by the linker (on program load)
    - Will increase performance during execution, but will slow down startup

ELF Sections

Lazy Binding

```
0000000000001020 <.plt>:
    1020:       push   QWORD PTR [rip+0x2fe2]        # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       jmp    QWORD PTR [rip+0x2fe4]        # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       nop    DWORD PTR [rax+0x0]

0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
    1036:       push   0x0
    103b:       jmp    1020 <.plt>

....

0000000000001135 <main>:
...
    114b:       call   1030 <puts@plt>
...

0000000000004000 <_GLOBAL_OFFSET_TABLE_>:
    4000:       f8 3d 00 00 00 00 00 00 00 00 00 00 00 00 00 00     .=..............
        ...
    4018:       36 10 00 00 00 00 00 00                             6.......
```

(1) The **`puts`** function is called. The function is on an external library, and it must be relocated.
So, it jumps to the **`puts@plt`**

ELF Sections

Lazy Binding

(2) At the PLT, the code doesn't jump to the final location, as it is not known (yet)

Instead, it jumps to an entry at the GOT (0x4018). In this case, the value is 0x1036, pointing to the code at line 8.

Remember: This is a static analysis, the dynamic linker is not working, so the symbol is unresolved

```
1  0000000000001020 <.plt>:
2      1020:       push   QWORD PTR [rip+0x2fe2]        # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
3      1026:       jmp    QWORD PTR [rip+0x2fe4]        # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
4      102c:       nop    DWORD PTR [rax+0x0]
5
6  0000000000001030 <puts@plt>:
7      1030:       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
8      1036:       push   0x0
9      103b:       jmp    1020 <.plt>
10
11 ....
12
13 0000000000001135 <main>:
14 ...
15     114b:       call   1030 <puts@plt>
16 ...
17
18 0000000000004000 <_GLOBAL_OFFSET_TABLE_>:
19     4000:       f8 3d 00 00 00 00 00 00 00 00 00 00 00 00 00     .=.............
20         ...
21     4018:       36 10 00 00 00 00 00 00                          6.......
```

ELF Sections

Lazy Binding

(3) A value 0 is pushed. This is an identifier that is stored to the stack. An index, actually.

The code then jumps to the .plt generic functions at 0x1020.

A new identifier is pushed (the address in the GOT that is missing the entry)

Code jumps to the Dynamic Linker

```
0000000000001020 <.plt>:
    1020:       push   QWORD PTR [rip+0x2fe2]        # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       jmp    QWORD PTR [rip+0x2fe4]        # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       nop    DWORD PTR [rax+0x0]

0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
    1036:       push   0x0
    103b:       jmp    1020 <.plt>

....

0000000000001135 <main>:
...
    114b:       call   1030 <puts@plt>
...

0000000000004000 <_GLOBAL_OFFSET_TABLE_>:
    4000:       f8 3d 00 00 00 00 00 00 00 00 00 00 00 00 00     .=.............
        ...
    4018:       36 10 00 00 00 00 00 00                         6.......
```

ELF Sections

Lazy Binding

(4) At the dynamic linker, it searches for the symbols in the mapped libraries and writes a value to the GOT at 0x4018.

Then he calls that address.

```
0000000000001020 <.plt>:
    1020:       push   QWORD PTR [rip+0x2fe2]        # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       jmp    QWORD PTR [rip+0x2fe4]        # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       nop    DWORD PTR [rax+0x0]

0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
    1036:       push   0x0
    103b:       jmp    1020 <.plt>

....

0000000000001135 <main>:
...
    114b:       call   1030 <puts@plt>
...

0000000000004000 <_GLOBAL_OFFSET_TABLE_>:
    4000:       f8 3d 00 00 00 00 00 00 00 00 00 00 00 00 00     .=.............
        ...
    4018:       36 10 00 00 00 00 00 00                         6.......
```

ELF Sections

Lazy Binding

(2.1) At the PLT, the code doesn't jump to the final location, as it is not known (yet)

Instead, it jumps to an entry at the GOT (**0x4018**).

If the program is executing, and it is the second time puts is called, the entry has **0x7ffff651910**, which points to the real puts.

This was obtained by loading the binary in GDB and using GEF

```
0000000000001020 <.plt>:
    1020:       push   QWORD PTR [rip+0x2fe2]        # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       jmp    QWORD PTR [rip+0x2fe4]        # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       nop    DWORD PTR [rax+0x0]

0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
    1036:       push   0x0
    103b:       jmp    1020 <.plt>

....

0000000000001135 <main>:
...
    114b:       call   1030 <puts@plt>
...
```

```
gef➤  got

GOT protection: Partial RelRO | GOT functions: 1

[0x8004018] puts@GLIBC_2.2.5  →  0x7ffff651910
```

ELF Sections

.rel.*, .rela.*

- Tables containing information to the dynamic linker about the required relocations
  - **R_X86_64_GLOB_DAT**: GOT offset should be filled with the symbol address (Lines 8-12)
  - **R_X86_64_JUMP_SLO**: Jump Slots to be represented in the **.got.plt** and **.plt** sections as shown previously (Line 16)

```
1  $ readelf --relocs hello
2
3  Relocation section '.rela.dyn' at offset 0x488 contains 8 entries:
4    Offset          Info           Type           Sym. Value    Sym. Name + Addend
5  000000003de8  000000000008 R_X86_64_RELATIVE                    1130
6  000000003df0  000000000008 R_X86_64_RELATIVE                    10f0
7  000000004028  000000000008 R_X86_64_RELATIVE                    4028
8  000000003fd8  000100000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTMClone + 0
9  000000003fe0  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
10 000000003fe8  000400000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
11 000000003ff0  000500000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCloneTa + 0
12 000000003ff8  000600000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0
13
14 Relocation section '.rela.plt' at offset 0x548 contains 1 entry:
15   Offset          Info           Type           Sym. Value    Sym. Name + Addend
16 000000004018  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

ELF Sections

.dynamic section

- Contains information instructing the operating system/dynamic linker to load the binary
  - Address of important tables
  - Flags
  - Required libraries
  - Debug flags
  - INIT/FINI addresses

```
$ readelf --dynamic hello

Dynamic section at offset 0x2df8 contains 26 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x1000
 0x000000000000000d (FINI)               0x11c4
 0x0000000000000019 (INIT_ARRAY)         0x3de8
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x3df0
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x308
 0x0000000000000005 (STRTAB)             0x3d8
 0x0000000000000006 (SYMTAB)             0x330
 0x000000000000000a (STRSZ)              130 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x4000
 0x0000000000000002 (PLTRELSZ)           24 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x548
 0x0000000000000007 (RELA)               0x488
 0x0000000000000008 (RELASZ)             192 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffb (FLAGS_1)            Flags: PIE
 0x000000006ffffffe (VERNEED)            0x468
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x45a
 0x000000006ffffff9 (RELACOUNT)          3
 0x0000000000000000 (NULL)               0x0
```

ELF Program Headers

Overview

- Provide a **segment view** of the binary, complementing the **section view**
  - Type of segment, offset in the binary file, alignments, virtual addresses to be considered
  - **Target the operating system** that will load the program **and not the linker** as the sections do

```
$ readelf --wide --segments hello

Elf file type is DYN (Shared object file)
Entry point 0x1050
There are 11 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x000268 0x000268 R   0x8
  INTERP         0x0002a8 0x00000000000002a8 0x00000000000002a8 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x000560 0x000560 R   0x1000
  LOAD           0x001000 0x0000000000001000 0x0000000000001000 0x0001cd 0x0001cd R E 0x1000
  LOAD           0x002000 0x0000000000002000 0x0000000000002000 0x000158 0x000158 R   0x1000
  LOAD           0x002de8 0x0000000000003de8 0x0000000000003de8 0x000248 0x000250 RW  0x1000
  DYNAMIC        0x002df8 0x0000000000003df8 0x0000000000003df8 0x0001e0 0x0001e0 RW  0x8
  NOTE           0x0002c4 0x00000000000002c4 0x00000000000002c4 0x000044 0x000044 R   0x4
  GNU_EH_FRAME   0x002010 0x0000000000002010 0x0000000000002010 0x00003c 0x00003c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x002de8 0x0000000000003de8 0x0000000000003de8 0x000218 0x000218 R   0x1
```

ELF Program Headers

Types

- **LOAD**: Segment should be loaded in memory
- **INTERP**: Segment containing the name of the interpreter to be used
- **DYNAMIC**: Segment containing the **`.dynamic`** section, to be used by the interpreter

```
$ readelf --wide --segments hello

Elf file type is DYN (Shared object file)
Entry point 0x1050
There are 11 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x000268 0x000268 R   0x8
  INTERP         0x0002a8 0x00000000000002a8 0x00000000000002a8 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x000560 0x000560 R   0x1000
  LOAD           0x001000 0x0000000000001000 0x0000000000001000 0x0001cd 0x0001cd R E 0x1000
  LOAD           0x002000 0x0000000000002000 0x0000000000002000 0x000158 0x000158 R   0x1000
  LOAD           0x002de8 0x0000000000003de8 0x0000000000003de8 0x000248 0x000250 RW  0x1000
  DYNAMIC        0x002df8 0x0000000000003df8 0x0000000000003df8 0x0001e0 0x0001e0 RW  0x8
  NOTE           0x0002c4 0x00000000000002c4 0x00000000000002c4 0x000044 0x000044 R   0x4
  GNU_EH_FRAME   0x002010 0x0000000000002010 0x0000000000002010 0x00003c 0x00003c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x002de8 0x0000000000003de8 0x0000000000003de8 0x000218 0x000218 R   0x1
```

Dynamic Linker

Dynamic Linker

- The dynamic linker is vital for the loading process, and can aid reversing a program
  - Provide information about the loaded libraries
  - Help debugging the linking process
  - Force linking
  - And many other
- Communication is achieved through Environmental Variables
  - In the format LD_*
  - Setting a variable, or setting a variable with a specific value, activates Linker features

Dynamic Linker

LD_LIBRARY_PATH

- A list of directories in which to search for ELF libraries at execution time.
  - The items in the list are separated by either colons or semicolons
  - A zero-length directory name indicates the current working directory.

- Activating: LD_LIBRARY_PATH=libs ./progname
  - Will make the linker look into ./libs while loading libraries for the program
  - Allows having a different set of libraries for the program (E.g., debug versions)

Dynamic Linker

LD_BIND_NOW

- Causes the dynamic linker to **resolve all symbols at program startup** instead of deferring function call resolution to the point when they are first referenced.
  - Especially useful for debug as all symbols point to their correct location

- Activated by setting the variable: `LD_BIND_NOW=1 progname`

```
gef➤ got

GOT protection: Partial RelRO | GOT functions: 4

[0x8004018] pthread_create@GLIBC_2.2.5 → 0x8001036
[0x8004020] printf@GLIBC_2.2.5 → 0x7fffff618560
[0x8004028] pthread_exit@GLIBC_2.2.5 → 0x8001056
[0x8004030] exit@GLIBC_2.2.5 → 0x8001066
```

`LD_BIND_NOT not set`

```
gef➤ got

GOT protection: Partial RelRO | GOT functions: 4

[0x8004018] pthread_create@GLIBC_2.2.5 → 0x7fffff797280
[0x8004020] printf@GLIBC_2.2.5 → 0x7fffff618560
[0x8004028] pthread_exit@GLIBC_2.2.5 → 0x7fffff7981d0
[0x8004030] exit@GLIBC_2.2.5 → 0x7fffff5f9ea0
```

`LD_BIND_NOW is set`

Dynamic Linker

LD_DEBUG

- Output verbose debugging information about the operation of the dynamic linker.
  - Allows tracing the operation of the linker
  - Debug where libraries are loading from
  - Determine if libraries are being loaded and which symbols trigger the event
  - Determine the search path used looking for libraries

- The content of this variable is one of more of the following categories, separated by colons/commas, spaces:
  - help, all, bindings, files, reloc, scopes, statistics, symbols, unused, version

- Use: `LD_DEBUG=option programname`

Dynamic Linker

LD_DEBUG

```
$ LD_DEBUG=all ./hello_thread
...

      7441:     relocation processing: /lib/x86_64-linux-gnu/libc.so.6 (lazy)
      7441:     symbol=_res;  lookup in file=./hello_thread [0]
      7441:     symbol=_res;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=_res;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `_res' [GLIBC_2.2.5]
      7441:     symbol=stderr;  lookup in file=./hello_thread [0]
      7441:     symbol=stderr;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=stderr;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `stderr' [GLIBC_2.2.5]
      7441:     symbol=error_one_per_line;  lookup in file=./hello_thread [0]
      7441:     symbol=error_one_per_line;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=error_one_per_line;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `error_one_per_line' [GLIBC_2.2.5]
      7441:     symbol=__morecore;  lookup in file=./hello_thread [0]
      7441:     symbol=__morecore;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=__morecore;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__morecore' [GLIBC_2.2.5]
      7441:     symbol=__key_encryptsession_pk_LOCAL;  lookup in file=./hello_thread [0]
      7441:     symbol=__key_encryptsession_pk_LOCAL;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=__key_encryptsession_pk_LOCAL;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__key_encryptsession_pk_LOCAL' [GLIBC_2.2.5]
      7441:     symbol=__libpthread_freeres;  lookup in file=./hello_thread [0]
      7441:     symbol=__libpthread_freeres;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libpthread.so.0 [0]: normal symbol `__libpthread_freeres'
      7441:     symbol=__progname_full;  lookup in file=./hello_thread [0]
      7441:     symbol=__progname_full;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=__progname_full;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
      7441:     binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__progname_full' [GLIBC_2.2.5]
      7441:     symbol=__ctype32_tolower;  lookup in file=./hello_thread [0]
      7441:     symbol=__ctype32_tolower;  lookup in file=/lib/x86_64-linux-gnu/libpthread.so.0 [0]
      7441:     symbol=__ctype32_tolower;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
```

Dynamic Linker

LD_PRELOAD

- A list of additional, user-specified, ELF shared objects to be loaded before all others.
  - This feature can be used to **selectively override functions** in other shared objects.
  - Symbols present in the provided ELF Shared objects are used instead of the original
  - Only the functions available in the shared object will be over written

- Use: LD_PRELOAD=./liboverride.so progname
  - Useful to provide custom implementations of any function in the program
  - Custom implementation can call the original implementation through manual symbol loading

hello_thread.c

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   pthread_exit(NULL);
}
```

hello_thread.c

Dynamic symbols

```
$ readelf --dyn-syms hello_thread

Symbol table '.dynsym' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND pthread_create@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND pthread_exit@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND exit@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     9: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
```

hello_thread.c

liboverride.c – compile with gcc -shared -fPIC -o liboverride.so liboverride.c -ldl

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <unistd.h>
#include <sys/types.h>

void pthread_exit(){
        void (*orig_pthread_exit)(void) = dlsym(RTLD_NEXT, "pthread_exit");

        printf("pthread_exit entry\n");
        orig_pthread_exit();
        printf("pthread_exit exit\n");
}

int pthread_create(void* a, void* b, void * c, void* d){
        int (*orig_pthread_create)(void*, void*, void*, void*) = dlsym(RTLD_NEXT, "pthread_create");
        printf("pthread_create entry: %p %p %p %p\n", a, b, c, d);
        int r = orig_pthread_create(a, b, c, d);
        printf("pthread_create exit: ret=%d", r);
        return r;
}
```

Manually load original function

Call original function

hello_thread.c

Left: standard execution, right: LD_PRELOAD overriding some functions

```
$ ./hello_thread
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

```
LD_PRELOAD=./liboverride.so ./hello_thread
In main: creating thread 0
pthread_create entry: 0x7ffff9f3b5b0 (nil) 0x7f861ef96165 (nil)
pthread_create exit: ret=0
In main: creating thread 1
pthread_create entry: 0x7ffff9f3b5b8 (nil) 0x7f861ef96165 0x1
Hello World! It's me, thread #0!
pthread_create exit: ret=0
In main: creating thread 2
pthread_create entry: 0x7ffff9f3b5c0 (nil) 0x7f861ef96165 0x2
Hello World! It's me, thread #1!
pthread_exit entry
pthread_create exit: ret=0
In main: creating thread 3
pthread_exit entry
Hello World! It's me, thread #2!
pthread_exit entry
pthread_create entry: 0x7ffff9f3b5c8 (nil) 0x7f861ef96165 0x3
pthread_create exit: ret=0
In main: creating thread 4
pthread_create entry: 0x7ffff9f3b5d0 (nil) 0x7f861ef96165 0x4
Hello World! It's me, thread #3!
pthread_exit entry
Hello World! It's me, thread #4!
pthread_exit entry
pthread_create exit: ret=0
pthread_exit entry
```