

华中科技大学
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



2.5 Linux ELF 可执行文件

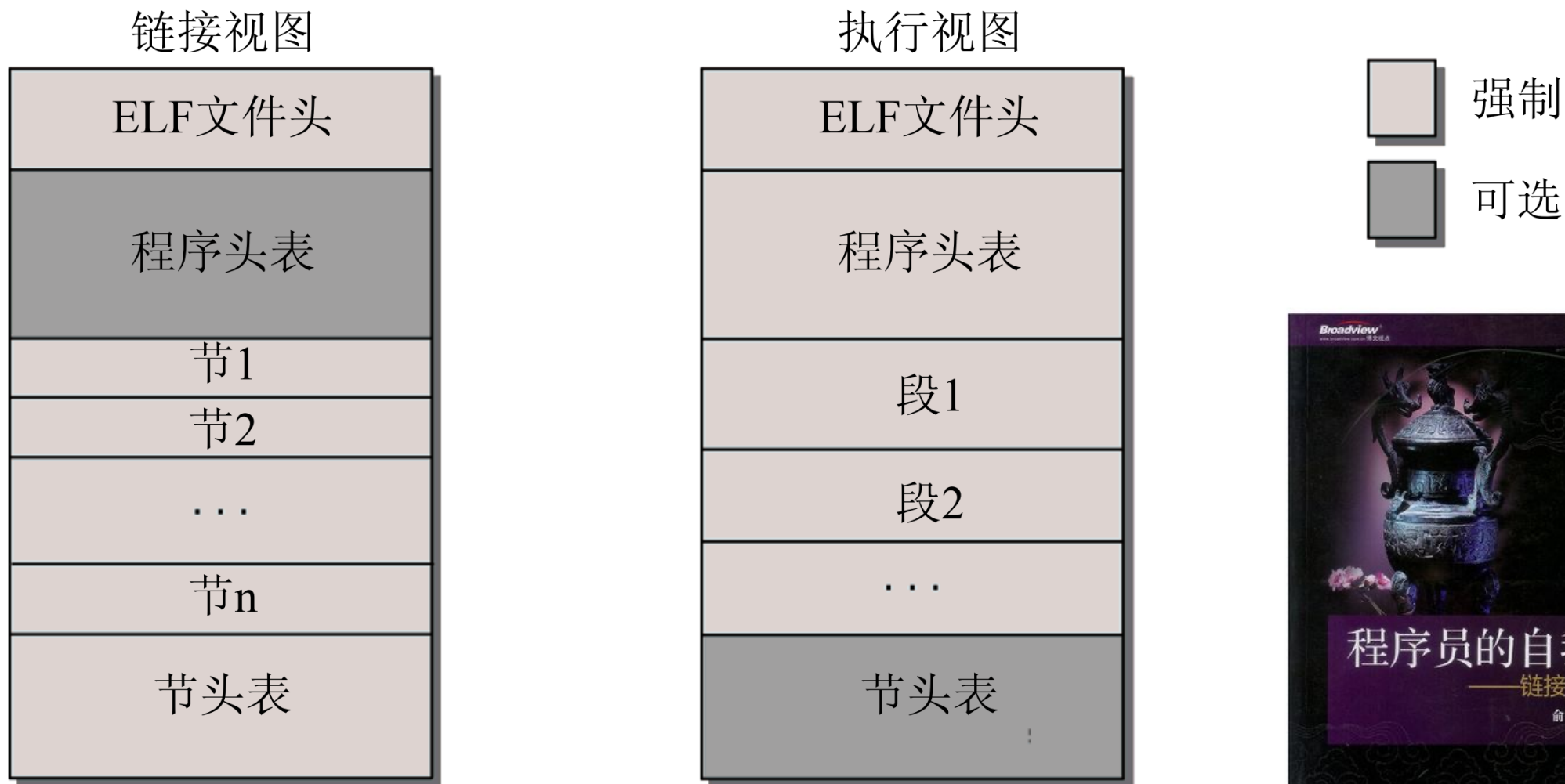
网络空间安全学院 慕冬亮

Email : dzm91@hust.edu.cn

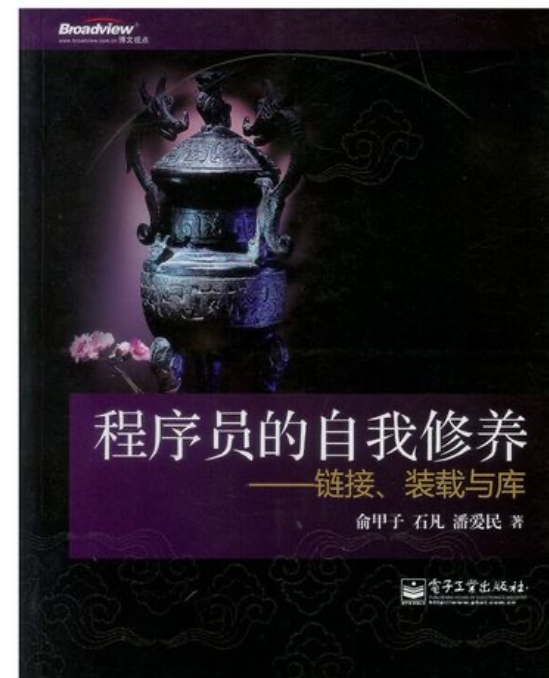
常用可执行文件格式

- Windows : PE (Portable Executable)
 - 目标文件: *.obj
 - 动态链接库: *.dll
 - 静态链接库: *.lib
- Linux/Unix : **ELF** (Executable Linkable Format)
 - 目标文件: *.o
 - 静态链接库: *.a
 - 动态链接库: *.so
- Mac OS X : Mach-O (Mach Object)
 - 目标文件: *.o
 - 静态链接库: *.a
 - 动态链接库: .dylib

Linux ELF 可执行文件



图E-1 ELF文件的基本布局



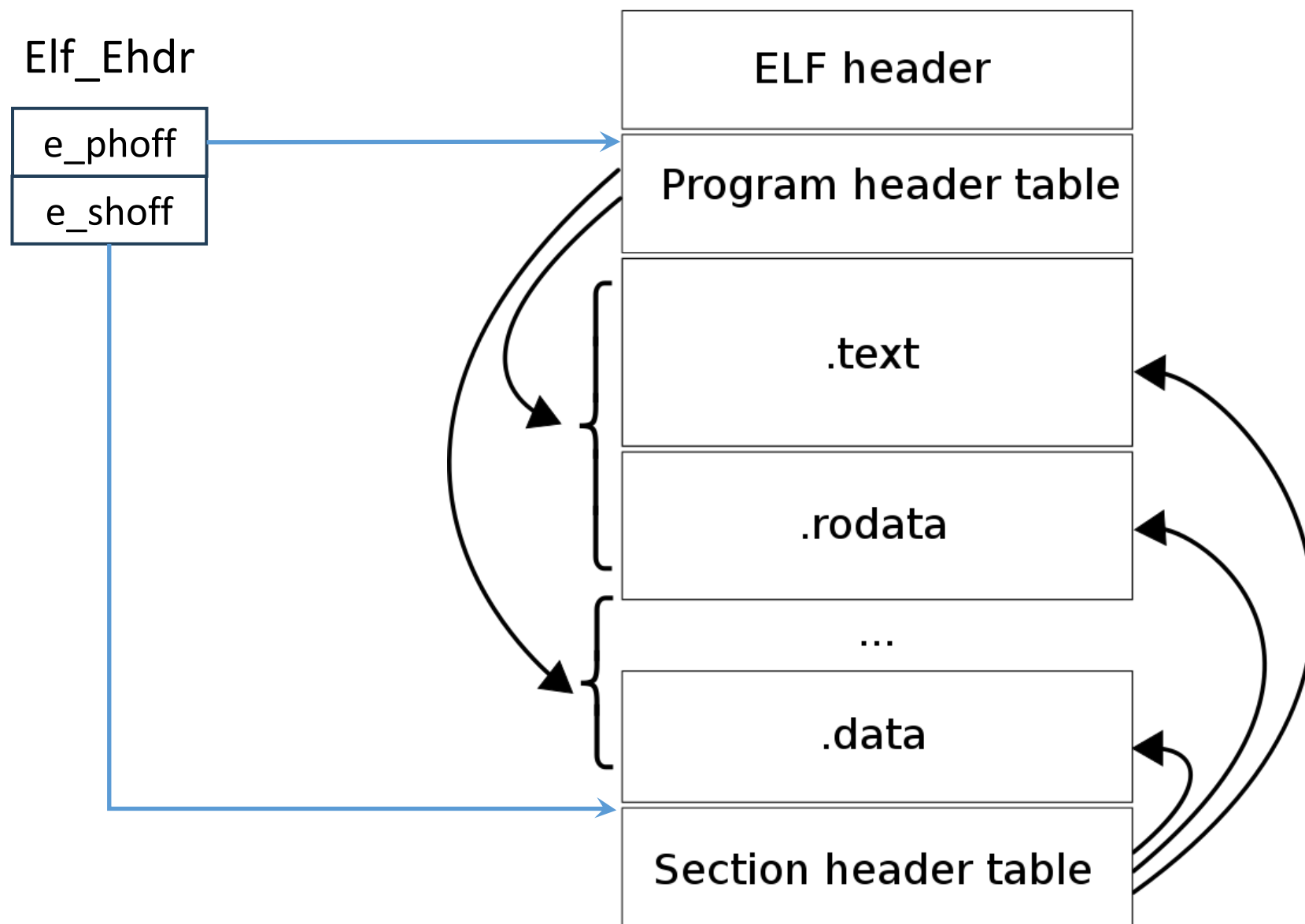
链接视图与执行视图

- 链接，不同于网页链接，是将不同目标文件组装为一个可执行文件的过程
- 执行，是将可执行文件映射到内存中并从入口点开始运行汇编代码的过程
- 链接视图：以节头表（Section Header Table）的角度来审视ELF文件，将不同目标文件的相同Section合并到一起
- 执行视图：以程序头表（Program Header Table）的角度审视ELF文件，将相同属性的节（如.data, .bss）组合成一个Segment，方便在执行过程中一起加载进入内存，减少磁盘拷贝操作

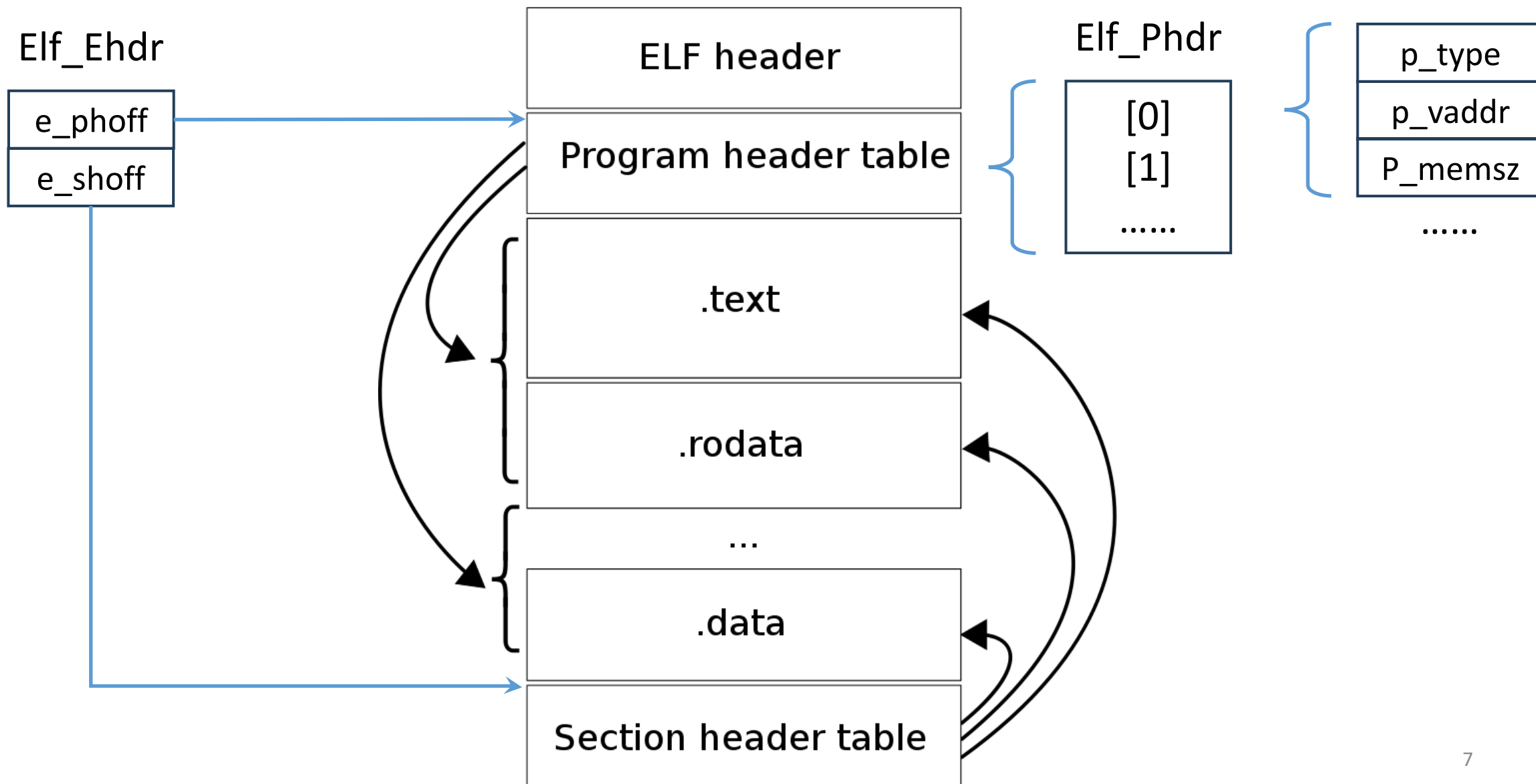
程序头表 vs 节头表

- Segment
 - ELF 文件执行时应该如何映射内存
 - 主要包含加载地址、文件中的偏移与长度、内存权限、对齐方式等信息
- Section
 - ELF 中具体各部分的功能，哪里是代码、哪里是数据，那些是符号信息
 - 主要包含 Section 类型、文件中的位置、带下等信息
 - 将不同的对象文件的代码数据信息合并，并修复互相引用
- Segment 与 Section 的关系
 - 相同权限的 Section 会放入同一个 Segment， 例如.text, .init和.fini

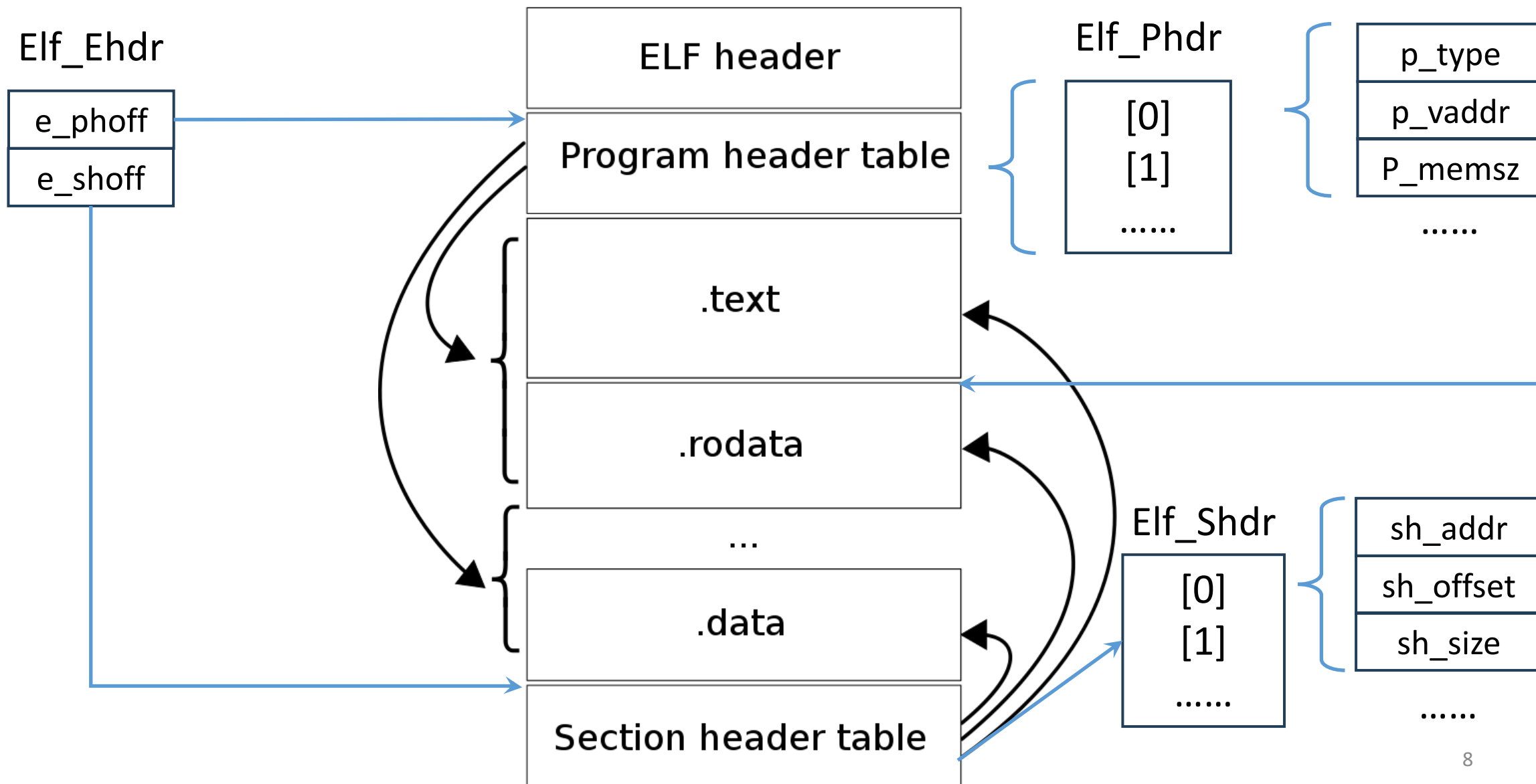
Linux ELF 可执行文件



Linux ELF 可执行文件



Linux ELF 可执行文件



ELF 文件头

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: EXEC (Executable file)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Entry point address: 0x401050
Start of program headers: 64 (bytes into file)
Start of section headers: 15784 (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: 34
Section header string table index: 33
```

```
typedef struct elf64_hdr {
    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;
```

ELF 程序头表

Section to Segment mapping:

Segment Sections...

00

01 .interp

02 .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym

.dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt

03 .init .plt .text .fini

04 .rodata .eh_frame_hdr .eh_frame

05 .init_array .fini_array .dynamic .got .got.plt .data .bss

06 .dynamic

07 .note.gnu.build-id .note.ABI-tag

08 .eh_frame_hdr

09

10 .init_array .fini_array .dynamic .got

DYNAMIC	0x00000000000002e20	0x000000000000403e20	0x000000000000403e20
	0x000000000000001d0	0x000000000000001d0	RW 0x8
NOTE	0x00000000000002c4	0x0000000000004002c4	0x0000000000004002c4
	0x0000000000000044	0x0000000000000044	R 0x4
GNU_EH_FRAME	0x00000000000002024	0x000000000000402024	0x000000000000402024
	0x000000000000004c	0x000000000000004c	R 0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x00000000000002e10	0x000000000000403e10	0x000000000000403e10
	0x000000000000001f0	0x000000000000001f0	R 0x1

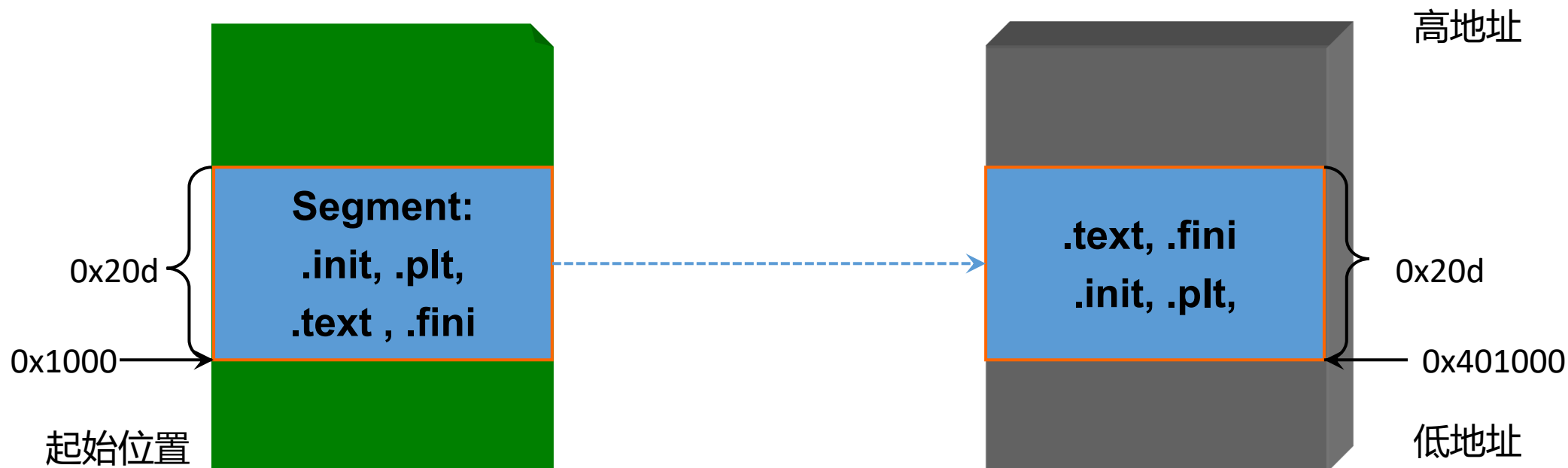
```
typedef struct elf64_phdr {
    Elf64_Word p_type;
    Elf64_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
} Elf64_Phdr;
```

ELF 程序头表

- `Elf64_Off p_offset;` `/* 段在文件中的偏移量 */`
- `Elf64_Addr p_vaddr;` `/* 段虚拟地址*/`
- `Elf64_Addr p_paddr;` `/* 段物理地址，在 x86 体系不起作用 */`
- `Elf64_Xword p_filesz;` `/* 文件中段的长度 */`
- `Elf64_Xword p_memsz;` `/* 内存中段的长度 */`

ELF 文件与虚拟内存之间映射

ELF 文件在执行过程中，会根据具有 LOAD 属性的Segment属性，进行数据映射



Type	Offset	VirtAddr	PhysAddr	
	FileSiz	MemSiz	Flags	Align
LOAD	0x0000000000000100	0x0000000000401000	0x0000000000401000	
	0x000000000000020d	0x000000000000020d	R E	0x1000

ELF 文件与虚拟内存之间映射

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
LOAD	0x0000000000000000	0x00000000000040000	0x00000000000040000
	0x00000000000000490	0x000000000000000490	R 0x1000
LOAD	0x00000000000001000	0x000000000000401000	0x000000000000401000
	0x0000000000000020d	0x00000000000000020d	R E 0x1000
LOAD	0x00000000000002000	0x000000000000402000	0x000000000000402000
	0x000000000000001b0	0x0000000000000001b0	R 0x1000
LOAD	0x00000000000002e10	0x000000000000403e10	0x000000000000403e10
	0x00000000000000228	0x000000000000000230	RW 0x1000
DYNAMIC	0x00000000000002e20	0x000000000000403e20	0x000000000000403e20
	0x000000000000001d0	0x0000000000000001d0	RW 0x8
NOTE	0x00000000000002c4	0x0000000000004002c4	0x0000000000004002c4
	0x00000000000000044	0x000000000000000044	R 0x4

ELF 节头表

There are 29 section headers, starting at offset 0x3910:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]	0000000000000000	NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0	0
[1]	.interp	PROGBITS	00000000004002a8	000002a8
	000000000000001c	0000000000000000	A 0 0	1
[2]	.note.gnu.bu[...]	NOTE	00000000004002c4	000002c4
	0000000000000024	0000000000000000	A 0 0	4
[3]	.note.ABI-tag	NOTE	00000000004002e8	000002e8
	0000000000000020	0000000000000000	A 0 0	4
[4]	.gnu.hash	GNU_HASH	0000000000400308	00000308
	000000000000001c	0000000000000000	A 5 0	8
[5]	.dynsym	DYNSYM	0000000000400328	00000328
	0000000000000078	0000000000000018	A 6 1	8
[6]	.dynstr	STRTAB	00000000004003a0	000003a0
	0000000000000045	0000000000000000	A 0 0	1
[7]	.gnu.version	VERSYM	00000000004003e6	000003e6
	000000000000000a	0000000000000002	A 5 0	2
[8]	.gnu.version_r	VERNEED	00000000004003f0	000003f0
	0000000000000020	0000000000000000	A 6 1	8
[9]	.rela.dyn	RELA	0000000000400410	00000410
	0000000000000030	0000000000000018	A 5 0	8
[10]	.rela.plt	RELA	0000000000400440	00000440
	0000000000000030	0000000000000018	AI 5 22	8
[11]	.init	PROGBITS	0000000000401000	00001000
	0000000000000017	0000000000000000	AX 0 0	4

```
typedef struct elf64_shdr {
    Elf64_Word sh_name;
    Elf64_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word sh_link;
    Elf64_Word sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

真正的数据和代码都是存在节中！

ELF 节头表

- `.bss` 保存程序未初始化的数据节，在程序开始运行前填充0字节
- `.data` 包含已经初始化的程序数据，在程序运行期间可更改
- `.rodata`保存了程序使用的只读数据，不能修改，例如字符串
- `.init`和`.fini`保存了程序初始化和结束时执行的机器指令，通常是由编译器及其辅助工具创建，为程序提供合适的运行时环境
- `.text`保存了主要的机器指令
- `.plt`包含所有外部函数调用的桩函数，将位置无关转移为绝对地址
- `.got`与`.got.plt`提供对外部共享库的访问入口，由动态链接器进行动态修改，分别包含全局变量的引用地址与外部函数的引用地址

ELF 节头表

- Elf64_Word sh_name; /* 节名，字符串表中的索引 */
- Elf64_Xword sh_flags; /* 节的各种属性 */
- Elf64_Addr sh_addr; /* 节在执行时的虚拟地址 */
- Elf64_Off sh_offset; /* 节在文件中的偏移量 */
- Elf64_Xword sh_size; /* 节长度，单位为字节 */

.shstrtab 包含了一个字符串表，定义了节名称

```
2E 73 74 72 74 61 62 00 2E 73 68 73 .strtab..shs
74 72 74 61 62 00 2E 69 6E 74 65 72 trtab..inter
70 00 2E 6E 6F 74 65 2E 67 6E 75 2E p..note.gnu.
62 75 69 6C 64 2D 69 64 00 2E 6E 6F build-id..no
74 65 2E 41 42 49 2D 74 61 67 00 2E te.ABI-tag..
67 6E 75 2E 68 61 73 68 00 2E 64 79 gnu.hash..dy
6E 73 79 6D 00 2E 64 79 6E 73 74 72 nsym..dynstr
00 2E 67 6E 75 2E 76 65 72 73 69 6F ..gnu.version
6E 00 2E 67 6E 75 2E 76 65 72 73 69 n..gnu.version
6F 6E 5F 72 00 2E 72 65 6C 61 2E 64 on_r..rela.d
79 6E 00 2E 72 65 6C 61 2E 70 6C 74 yn..rela.plt
00 2E 69 6E 69 74 00 2E 74 65 78 74 ..init..text
00 2E 66 69 6E 69 00 2E 72 6F 64 61 ..fini..rodata
74 61 00 2E 65 68 5F 66 72 61 6D 65 ta..eh_frame
5F 68 64 72 00 2E 65 68 5F 66 72 61 _hdr..eh_frame
6D 65 00 2E 69 6E 69 74 5F 61 72 72 me..init_array
61 79 00 2E 66 69 6E 69 5F 61 72 72 ay..fini_array
61 79 00 2E 64 79 6E 61 6D 69 63 00 ay..dynamic.
2E 67 6F 74 00 2E 67 6F 74 2E 70 6C .got..got.plt
74 00 2E 64 61 74 61 00 2E 62 73 73 t..data..bss
```


ELF 文件与虚拟内存的映射

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info	Align
[13]	.text	PROGBITS	0000000000401050	00001050
	000000000000001c1	0000000000000000	AX 0 0	16

ELF CrakeMe

0 / 8

Level 2.0

1. 通过反汇编工具查看 .text 段的逻辑，确定要修改的汇编语句
2. 根据反汇编工具显示的虚拟地址位置(如40116d)计算该指令在文件中的偏移地址；

$$X - 0x1050 = 0x40116d - 0x401050$$

计算可得: $X = 0x116d$

因为ELF文件相应 Segment 加载过程未发生膨胀或缩减

动态链接

- 为什么需要动态链接？
 - 内存和磁盘空间的浪费
 - 程序开发，更新，部署和发布
 - 程序可扩展性和兼容性
- 动态链接的缺点？
 - 静态链接要比动态库快约1~5%
 - 外部数据与函数调用需要GOT定位
 - 动态链接在运行时完成，**只需一次**

```
401136:      48 8d 3d c7 0e 00 00      lea    rdi,[rip+0xec7]
40113d:      e8 ee fe ff ff          call   401030 <puts@plt>
```

```
0000000000401030 <puts@plt>:
401030:      ff 25 e2 2f 00 00      jmp     QWORD PTR [rip+0x2fe2]
401036:      68 00 00 00 00          push    0x0
40103b:      e9 e0 ff ff ff          jmp     401020 <.plt>
```

404018
puts@GLIBC_2.2.5

```
[22] .got.plt      PROGBITS      0000000000404000 00003000
      0000000000000028 0000000000000008 WA          0      0      8
```

动态链接

Hex dump of section '.got.plt':

NOTE: This section has relocations against it,

```
0x00404000 203e4000 00000000 00000000 00000000
0x00404010 00000000 00000000 36104000 00000000
0x00404020 46104000 00000000
```

0x401036

0000000000401030 <puts@plt>:

```
401030: ff 25 e2 2f 00 00    jmp     QWORD PTR [rip+0x2fe2]
401036: 68 00 00 00 00      push    0x0
40103b: e9 e0 ff ff ff      jmp     401020 <.>plt>
```

0000000000401020 <.>plt>:

```
401020: ff 35 e2 2f 00 00    push    QWORD PTR [rip+0x2fe2] GOT[1]
401026: ff 25 e4 2f 00 00    jmp     QWORD PTR [rip+0x2fe4] GOT[2]
40102c: 0f 1f 40 00          nop     DWORD PTR [rax+0x0]
```

GOT[2] = `_dl_runtime_resolve`地址，调用后可找到具体符号(puts)的地址，并写入对应的GOT项

动态链接图示

/*一开始没有重定位的时候将 puts@got 填成 lookup_puts的地址*/

```
void puts@plt()
```

```
{
```

```
address_good:
```

```
    jmp *puts@got
```

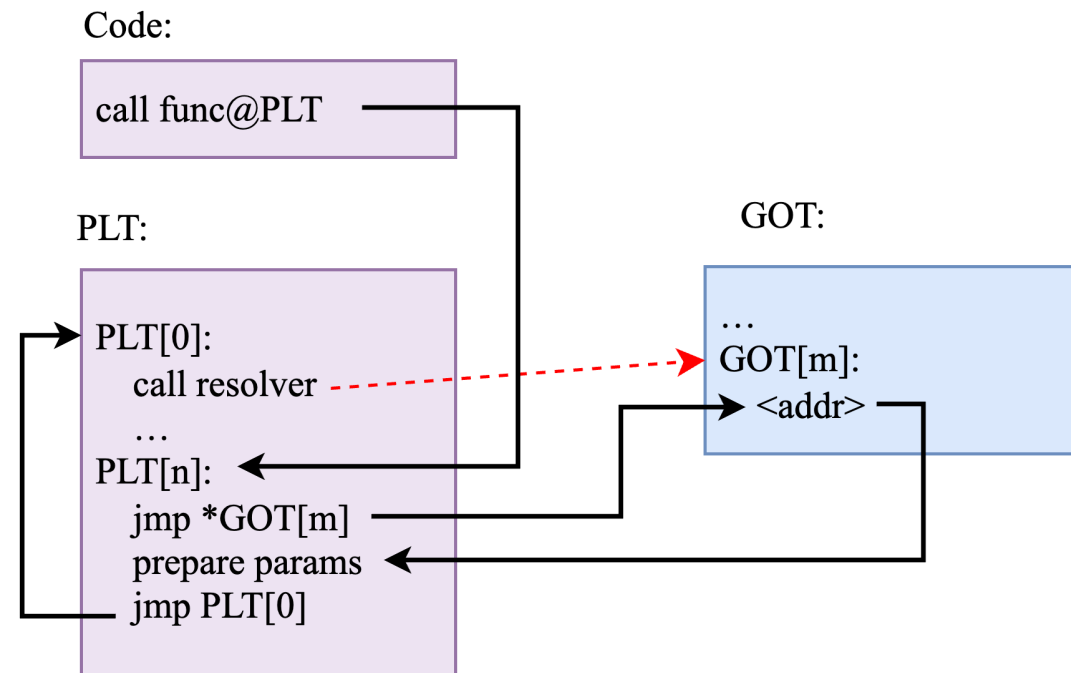
```
lookup_puts:
```

```
    调用_dl_runtime_resolve查找puts,
```

```
    并写回到 printf@got
```

```
    goto address_good;
```

```
}
```



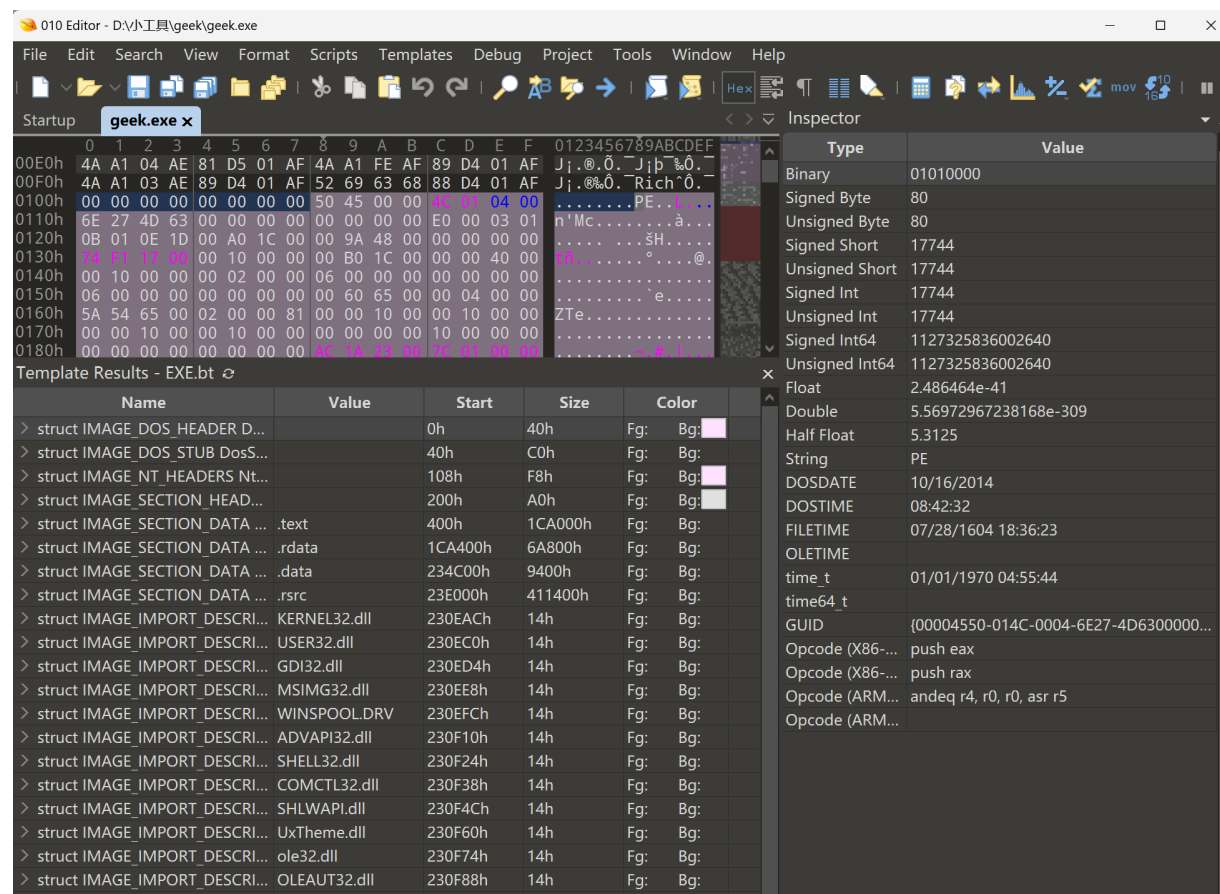
动态链接总结

- 需要进行动态链接的ELF可执行程序在编译时会自动生成DYNAMIC段、GOT表和PLT表；
- 对动态库的每个函数调用都会在GOT(从GOT[3]开始)和PLT(从第二段汇编指令开始)中生成一项；
- 解析器在获得控制权后会在GOT[1]和GOT[2]放置解析函数的参数和入口地址；PLT的第一段汇编指令会将GOT[1]元素压栈并跳转到GOT[2]指定的函数位置；
- 进行过一次动态调用后，GOT中对应的元素中就记录了库函数的实际加载地址，后续的调用就可以进行直接跳转。

逆向分析实例 - 相关工具简介

- 010 Editor

- 查看并编辑任何二进制文件和文本文件，包括 EXE、DLL、PDF、图片文件、音频文件等
- 其独特的二进制模板可以解析任何二进制文件格式
- 具有查找、替换、二进制比较、校验和/散列算法、直方图等分析和编辑功能



相关工具简介

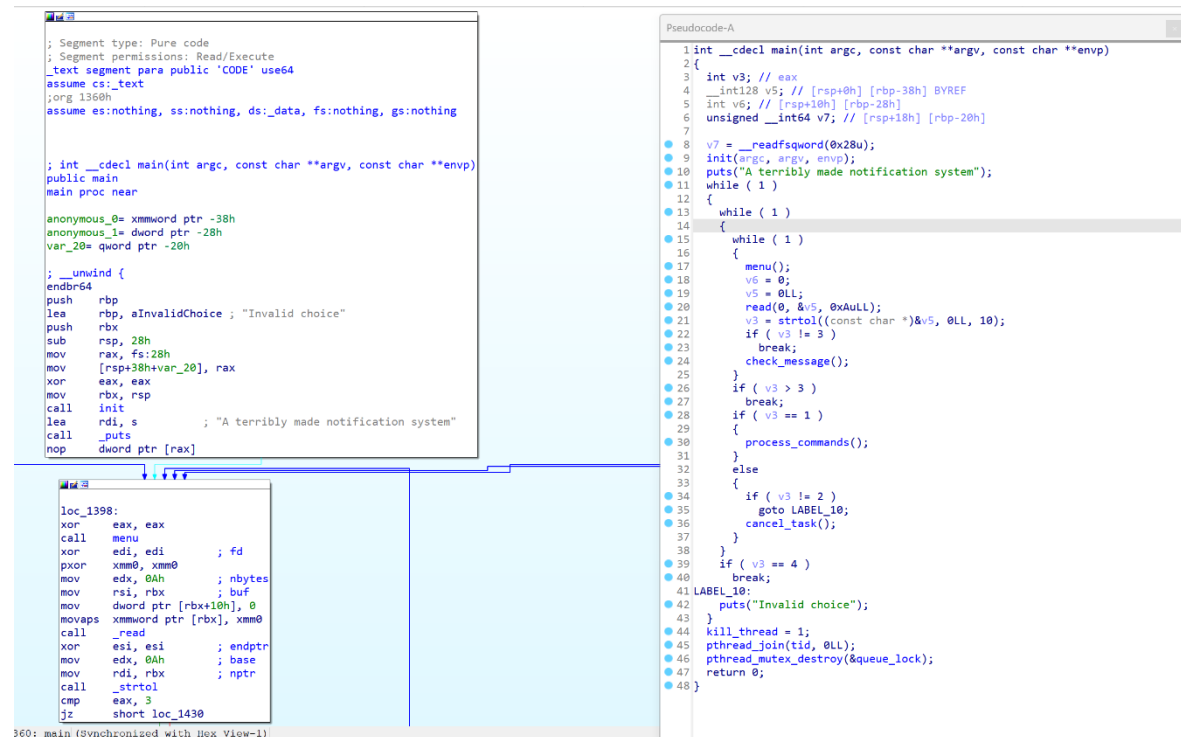
- Objdump 用于显示二进制目标文件的信息
 - -d, --disassemble : 反汇编可执行的 section 内容
 - -D, --disassemble-all : 反汇编所有 section 内容
 - -f, --file-headers : 显示文件头信息
 - -h, --[section-]headers : 显示目标文件各个 section 的头部摘要信息
 - -s, --full-contents : 显示指定 section 的完整内容。默认所有的非空 section 都会被显示
 - -S, --source : 尽可能反汇编出源代码, 尤其当编译的时候指定了 -g 这种调试参数时, 效果比较明显。隐含了 -d 参数
 - -t, --syms : 显示文件的符号表入口。类似于 nm -s 提供的信息
 - -x, --all-headers : 显示所有可用的头信息, 包括符号表、重定位入口。-x 等价于 -a -f -h -r -t 同时指定
- Hexedit 二进制编辑器, 可以同时显示文件的十六进制和 ASCII 视图
- Readelf 可以显示关于 ELF 格式文件内容的信息
- Hexdump 用于查看二进制文件
- Hexyl 是一个终端的十六进制查看器, 使用彩色输出来区分不同类别的字节



相关工具简介

- 静态反汇编工具：IDA Pro

- 当前最强大的静态反汇编软件（也能进行简单的动态调试）
- 能将庞大的汇编指令序列分成不同层次的单元、模块、函数，并给予标注和注解，以便交叉引用
- 能自动识别和标注 VC、BC、TC、Delphi 等常用编译器的标准库函数
- 能以图形化的方式显示函数内部的执行流程
- 可以将标注好的函数名、注释等信息导出为 map 文件，供 OllyDbg 动态调试时使用



The screenshot displays the IDA Pro interface with three main panels. The top-left panel shows assembly code for a function named `main`, including segment declarations and instructions like `push rbp`, `lea rbp, aInvalidChoice`, and `call _init`. The bottom-left panel shows a function graph for `loc_1398`, illustrating the control flow between instructions. The right panel shows the corresponding pseudocode, which includes variable declarations, initialization, and a loop structure with conditional branches and function calls like `menu()` and `process_commands()`.



<https://hex-rays.com/ida-free/>

相关工具简介

- 动态调试工具

- SoftICE (Soft In Circuit Emulator)

- 工作在操作系统的 Ring 0，以软件的方式实现了监视 CPU 的所有动作，可以调试驱动等内核对象，也可使用RCP/IP连接进行远程调试
 - 暴力中断所有进程，不如 OllyDbg 使用方便
 - 所有功能通过调试命令完成，并且有可能无意间修改系统很底层的东西，无经验者使用经常出现死机、蓝屏
 - ctrl + d: 呼出调试界面（但不易对界面截图）

- GDB

- Linux 下功能全面的调试工具
 - 支持对多种编程语言的调试，包括C、C++、Go、Objective-C、Rust等
 - 调试功能强大，支持断点、单步执行、查看变量、查看寄存器、查看堆栈等手段
 - 多种增强插件，pwndbg, gef, peda, pwngdb等

相关工具简介

- Ghidra

- Ghidra是一个软件逆向工程框架
- 支持Windows, MacOS和Linux等各种平台上分析二进制文件
- 功能强大, 包括反汇编、汇编、反编译、绘图和脚本, 以及数百个其他功能
- 支持多种处理器指令集和可执行格式, 并且可以在用户交互和自动化模式下运行
- 用户还可以使用 Java 或 Python 开发自己的 Ghidra 扩展组件和/或脚本

- Binary Ninja

- Ninja 也是一款逆向工程工具
- 支持在Windows、macOS 和 Linux上反汇编可执行文件
- 支持将代码反编译为 C 或 BNIL 以适应任何多种体系架构
- 支持在任何受支持的体系结构或平台上本地或远程调试程序
- 提供了多种方式来修改二进制文件, 可以直接编辑十六进制, 也可以使用内置的C编译器直接书写C代码来进行操作

ELF 二进制破解演示

Magic Number and File Header

ELF 二进制破解演示

Program header or segments

ELF 二进制破解演示

Section headers

ELF 二进制破解演示

objdump

ELF 二进制破解演示

Modify .text

精彩内容下章继续…

❖ 下堂课见

