

《Linux 系统分析》大作业 说明书

题 目 对 Linux 操作系统下 ELF 文件的分析与研究

摘 要

可执行文件是操作系统中最重要的文件类型之一,因为它们是实现操作的真正执行者。可执行文件的大小、运行速度、可扩展性和可移植性等与文件格式的定义紧密相关。因此对操作系统下的可执行文件格式分析研究是十分必要的。

本次作业要求在 Linux 中修改一个现有的 elf 可执行程序。让该程序运行后先执行一个特别的附件功能(附加功能是:创建或者打开一个指定文件写入 helloworld 的字符串)后再继续运行该程序,否则程序结束。注意:附加功能嵌入到原来的程序中,不是一个独立的程序。

1 ELF 文件介绍

1.1 ELF 文件格式

1.1.1 ELF 文件的类型

ELF 文件主要有三种类型:

(1) 可重定位文件包含了代码和数据,可与其它 ELF 文件建立一个可执行或共享的文件;

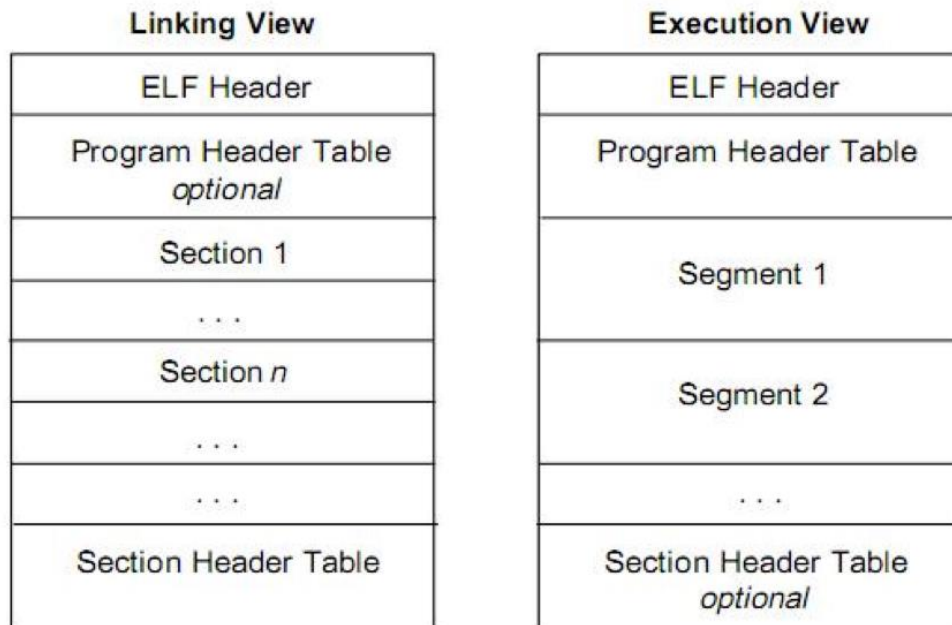
(2) 可执行文件时可直接执行的程序;

(3) 共享目标文件包括代码和数据,可以在两个地方链接。第一,连接器可以把它和其它可重定位文件和共享文件一起处理以建立另一个 ELF 文件;第二,动态链接器把它和一个可执行文件和其它共享文件结合在一起建立一个进程映像。

1.1.2 ELF 文件的组织

ELF 文件参与程序的连接(建立一个程序)和程序的执行(运行一个程序),编译器和链接器将其视为节头表(section headertable)描述的一些节(section)的集合,而加载器则将其视为程序头表(program header table)描述的段(segment)的集合,通常一个段可以包含多个节。可重定位文件都包含一个节头表,可执行文件都包含一个程序头表。共享文件两者都包含有。为此,ELF 文件格式同时提供了两种看待文件内容的方式,反映了不同行为的不同要求。

下图显示了 ELF 文件的组织。



1.2 深入探究

① ELF 所用的数据结构，32 位和 64 位所占的字节大小是不一样的。

```
wangsisl@wangsisl-virtual-machine: ~
File Edit View Search Terminal Help

#include <stdint.h>

/* Type for a 16-bit quantity. */
typedef uint16_t Elf32_Half;
typedef uint16_t Elf64_Half;

/* Types for signed and unsigned 32-bit quantities. */
typedef uint32_t Elf32_Word;
typedef int32_t  Elf32_Sword;
typedef uint32_t Elf64_Word;
typedef int32_t  Elf64_Sword;

/* Types for signed and unsigned 64-bit quantities. */
typedef uint64_t Elf32_Xword;
typedef int64_t  Elf32_Sxword;
typedef uint64_t Elf64_Xword;
typedef int64_t  Elf64_Sxword;

/* Type of addresses. */
typedef uint32_t Elf32_Addr;
typedef uint64_t Elf64_Addr;

/* Type of file offsets. */
typedef uint32_t Elf32_Off;
typedef uint64_t Elf64_Off;

/* Type for section indices, which are 16-bit quantities. */
typedef uint16_t Elf32_Section;
typedef uint16_t Elf64_Section;

/* Type for version symbol information. */
typedef Elf32_Half Elf32_Versym;
typedef Elf64_Half Elf64_Versym;
```

② ELF 文件头（ELF header）

进入终端输入：`cd /usr/include/elf.h`，查看 ELF 的文件头包含整个文件的控制结构。根据文件类型分为 32 位和 64 位，如图 1，图 2 所示。

```
wanglisi@wanglisi-virtual-machine: ~
File Edit View Search Terminal Help
/* The ELF file header. This appears at the start of every ELF file. */

#define EI_NIDENT (16)

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf32_Half e_type; /* Object file type */
    Elf32_Half e_machine; /* Architecture */
    Elf32_Word e_version; /* Object file version */
    Elf32_Addr e_entry; /* Entry point virtual address */
    Elf32_Off e_phoff; /* Program header table file offset */
    Elf32_Off e_shoff; /* Section header table file offset */
    Elf32_Word e_flags; /* Processor-specific flags */
    Elf32_Half e_ehsize; /* ELF header size in bytes */
    Elf32_Half e_phentsize; /* Program header table entry size */
    Elf32_Half e_phnum; /* Program header table entry count */
    Elf32_Half e_shentsize; /* Section header table entry size */
    Elf32_Half e_shnum; /* Section header table entry count */
    Elf32_Half e_shstrndx; /* Section header string table index */
} Elf32_Ehdr;
```

图 1 32 位的 elf 文件头数据结构

```
wanglisi@wanglisi-virtual-machine: ~
File Edit View Search Terminal Help

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf64_Half e_type; /* Object file type */
    Elf64_Half e_machine; /* Architecture */
    Elf64_Word e_version; /* Object file version */
    Elf64_Addr e_entry; /* Entry point virtual address */
    Elf64_Off e_phoff; /* Program header table file offset */
    Elf64_Off e_shoff; /* Section header table file offset */
    Elf64_Word e_flags; /* Processor-specific flags */
    Elf64_Half e_ehsize; /* ELF header size in bytes */
    Elf64_Half e_phentsize; /* Program header table entry size */
    Elf64_Half e_phnum; /* Program header table entry count */
    Elf64_Half e_shentsize; /* Section header table entry size */
    Elf64_Half e_shnum; /* Section header table entry count */
    Elf64_Half e_shstrndx; /* Section header string table index */
} Elf64_Ehdr;
```

图 2 64 位的 elf 文件头数据结构

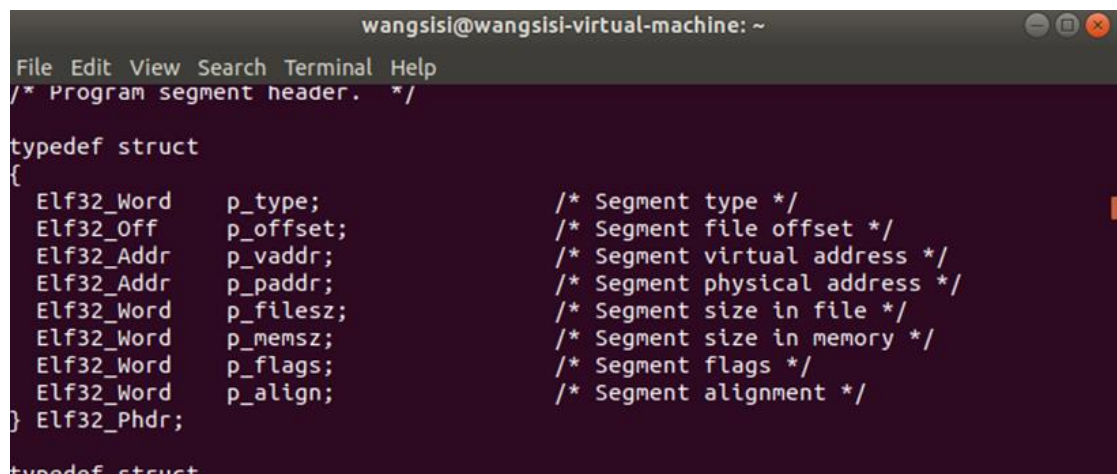
e_ident	标识 elf 文件
e_type	目标文件类型
e_machine	硬件平台
e_version	目标文件版本
e_entry	程序进入点
e_phoff	程序头表偏移量
e_shoff	节头表偏移量
e_flags	处理器特定标志
e_ehsize	elf 头部长度
e_phentsize	程序头表中的一个条目的长度
e_phnum	程序头表条目数目
e_shentsize	节头表中的一个条目的长度
e_shnum	节头表条目个数
e_shstrndx	节头表字符索引

表 1 文件头结构具体含义

Elf 头在程序的开始部位，作为引路表描述整个 ELF 的文件结构，其信息大致分为四部分：一是系统相关信息，二是目标文件类型，三是加载相关信息，四是链接相关信息。其中系统相关信息包括 elf 文件魔数(标识 elf 文件)，平台位数，数据编码方式，elf 头部版本，硬件平台 e_machine，目标文件版本 e_version，处理器特定标志 e_flags；这些信息的引入极大增强了 elf 文件的可移植性,使交叉编译成为可能。目标文件类型用 e_type 的值表示，可重定位文件为 1，可执行文件为 2，共享文件为 3；加载相关信息有：程序进入点 e_entry，程序头表偏移量 e_phoff，elf 头部长度 e_eh-size，程序头表中一个条目的长度 e_phentsize，程序头表条目数目 e_phnum；链接相关信息有：节头表偏移量 e_shoff，节头表中一个条目的长度 e_shentsize，节头表条目个数 e_shnum，节头表字符索引 e_shstrndx。可使用 readelf -h filename 来察看文件头的内容。

③程序头表(program header table)

Program header 所使用的数据结构如图 3，图 4 所示。

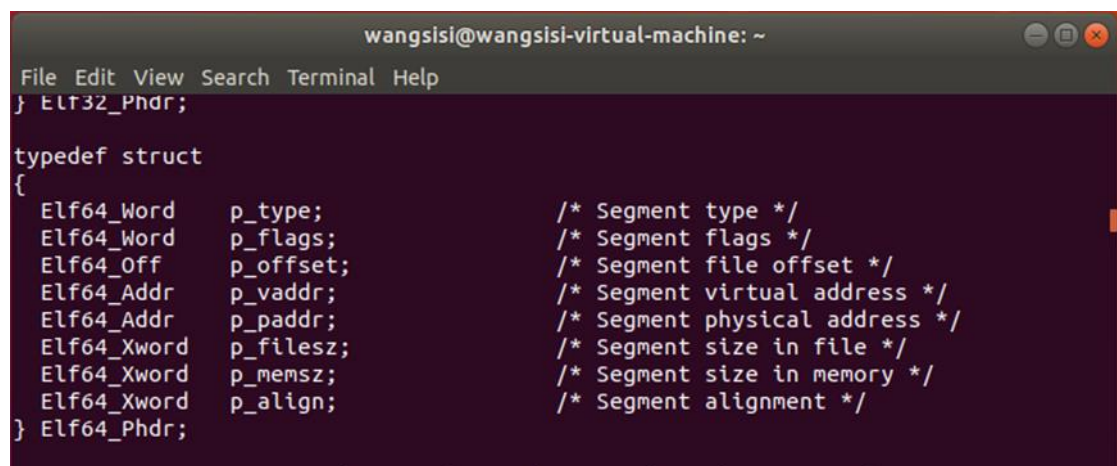


```
wanglisi@wanglisi-virtual-machine: ~
File Edit View Search Terminal Help
/* Program segment header. */

typedef struct
{
    Elf32_Word    p_type;           /* Segment type */
    Elf32_Off     p_offset;         /* Segment file offset */
    Elf32_Addr    p_vaddr;         /* Segment virtual address */
    Elf32_Addr    p_paddr;         /* Segment physical address */
    Elf32_Word    p_filesz;        /* Segment size in file */
    Elf32_Word    p_memsz;         /* Segment size in memory */
    Elf32_Word    p_flags;         /* Segment flags */
    Elf32_Word    p_align;         /* Segment alignment */
} Elf32_Phdr;

typedef struct
```

图 3 programe header 数据结构 (32 位)



```
wanglisi@wanglisi-virtual-machine: ~
File Edit View Search Terminal Help
} Elf32_Phdr;

typedef struct
{
    Elf64_Word    p_type;           /* Segment type */
    Elf64_Word    p_flags;         /* Segment flags */
    Elf64_Off     p_offset;         /* Segment file offset */
    Elf64_Addr    p_vaddr;         /* Segment virtual address */
    Elf64_Addr    p_paddr;         /* Segment physical address */
    Elf64_Xword   p_filesz;        /* Segment size in file */
    Elf64_Xword   p_memsz;         /* Segment size in memory */
    Elf64_Xword   p_align;         /* Segment alignment */
} Elf64_Phdr;
```

图 4 programe header 数据结构 (64 位)

以下为具体含义。

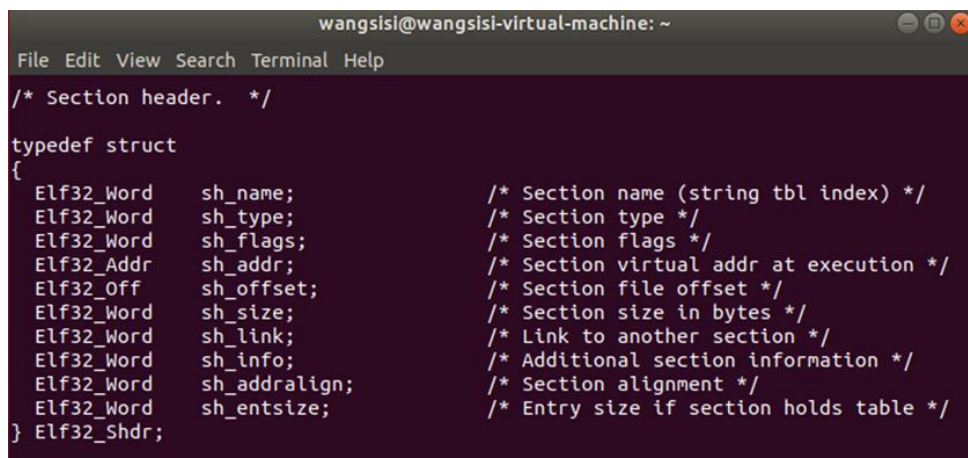
p_type	段的类型
p_offset	段的位置相对于文件开始处的偏移
p_vaddr	段在内存中的首字节地址
p_paddr	段的物理地址
p_filesz	段在文件映像中的字节数
p_memsz	段在内存映像中的字节数
p_flags	段的标记
p_align	段在内存中的对齐标记

表 2 程序表头结构具体含义

程序头表告诉系统如何建立一个进程映像，它是从加载执行的角度来看待 elf 文件，从它的角度看，elf 文件被分成许多段，elf 文件中的代码、链接信息和注释都以段的形式存放。每个段都在程序头表中有一个表项描述，包含以下属性：段的类型，段的驻留位置相对于文件开始处的偏移，段在内存中的首字节地址，段的物理地址，段在文件映像中的字节数，段在内存映像中的字节数，段在内存和文件中的对齐标记。可用 `readelf - l filename` 察看程序头表中的内容。

④节头表(section header table)

图 6，图 7 是节头表对应的结构。

A screenshot of a terminal window with a dark background. The title bar reads 'wangsis@wangsis-virtual-machine: ~'. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal displays the C code for the 'Section header' struct, starting with a comment '/* Section header. */' and a 'typedef struct' declaration. The struct contains fields for name, type, flags, address, offset, size, link, info, alignment, and entry size, each with a corresponding comment. The struct is named 'Elf32_Shdr' and is enclosed in curly braces.

```
/* Section header. */
typedef struct
{
    Elf32_Word    sh_name;           /* Section name (string tbl index) */
    Elf32_Word    sh_type;           /* Section type */
    Elf32_Word    sh_flags;          /* Section flags */
    Elf32_Addr    sh_addr;           /* Section virtual addr at execution */
    Elf32_Off     sh_offset;         /* Section file offset */
    Elf32_Word    sh_size;           /* Section size in bytes */
    Elf32_Word    sh_link;           /* Link to another section */
    Elf32_Word    sh_info;           /* Additional section information */
    Elf32_Word    sh_addralign;      /* Section alignment */
    Elf32_Word    sh_entsize;       /* Entry size if section holds table */
} Elf32_Shdr;
```

图 6 section 结构（32 位）


```
wangsis@wangsis-virtual-machine: ~  
File Edit View Search Terminal Help  
typedef struct  
{  
    Elf64_Word  sh_name;          /* Section name (string tbl index) */  
    Elf64_Word  sh_type;          /* Section type */  
    Elf64_Xword sh_flags;         /* Section flags */  
    Elf64_Addr  sh_addr;          /* Section virtual addr at execution */  
    Elf64_Off   sh_offset;        /* Section file offset */  
    Elf64_Xword sh_size;          /* Section size in bytes */  
    Elf64_Word  sh_link;          /* Link to another section */  
    Elf64_Word  sh_info;          /* Additional section information */  
    Elf64_Xword sh_addralign;     /* Section alignment */  
    Elf64_Xword sh_entsize;       /* Entry size if section holds table */  
} Elf64_Shdr;
```

图 7 section 结构（64 位）

sh_name	小节名在字符表中的索引
sh_type	小节的类型
sh_flags	小节属性
sh_addr	小节在运行时的虚拟地址
sh_offset	小节的文件偏移
sh_size	小节的大小，以字节为单位
sh_link	链接另外一小节的索引
sh_info	附加的小节信息
sh_addralign	小节的对齐
sh_entsize	一些 sections 保存着一张固定大小入口的表，就像符号表

表 3 节头表结构的具体含义

节头表描述程序节，为编译器和链接器服务。它把 elf 文件分成了许多节，每个节保存着用于不同目的的数据，这些数据可能被前面的程序头重复使用，完成一次任务所需的信息往往被分散到不同的节里。由于节中数据的用途不同，节被分成不同的类型，每种类型的节都有自己组织数据的方式。每一个节在节头表中都有一个表项描述该节的属性，节的属性包括小节名在字符表中的索引，类型，属性，运行时的虚拟地址，文件偏移，以字节为单位的大小，小节的对齐等信息，可使用 `readelf - S filename` 来察看节头表的内容。

2 ELF 文件解析

2.1 解析 ELF 头文件

首先，使用 `readelf` 工具查看 ELF 头文件的具体样式，如图 2-1 所示。这里我使用 64 位文件来展示我的思考过程。

```
wangsysi@wangsysi-virtual-machine: ~
File Edit View Search Terminal Help
wangsysi@wangsysi-virtual-machine:~$ 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:                0x530
  Start of program headers:           64 (bytes into file)
  Start of section headers:          6448 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           9
  Size of section headers:            64 (bytes)
  Number of section headers:          29
  Section header string table index: 28
```

图 2-1 readelf 工具查看头文件

C 语言解析得出的图如 2-2 所示。

```
E:\linux\大作业\新建文件夹\1_64.exe
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      : Shared object file
  Machine   : AMD X86-64 architecture
  version   : 0x1
  Entry point address : 0x530
  Start of program headers : 64 (bytes into file)
  Start of section headers : 6448 (bytes into file)
  Flags      : 0x0
  Size of this header      : 64 (bytes)
  Size of program headers  : 56 (bytes)
  Number of program headers : 9
  Size of section headers  : 64 (bytes)
  Number of section headers : 29
  Section header string table index: 28

-----
Process exited after 0.3088 seconds with return value 0
请按任意键继续. . .
```

图 2-2 自写代码运行结果

使用 C 语言开始对目标文件（可执行文件）解析。首先解析 Magic num，对应的就是 e_ident。如图 2-2 所示，它占 16 个字节，所以从 0 开始指针遍历到 16 输出。

```
/* The ELF file header. This appears
#define EI_NIDENT (16)

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /*
```

图 2-3 Magic

如图 2-4，前 4 个字节是 ELF 的 Magic Number，固定为 7f 45 4c 46。第 5 个字节指明 ELF 文件是 32 位还是 64 位的。第 6 个字节指明了数据的编码方式，即我们通常说的 little endian 或是 big endian。little endian。第 7

个字节指明了 ELF header 的版本号，目前值都是 1。第 8-16 个字节，都填充为 0。

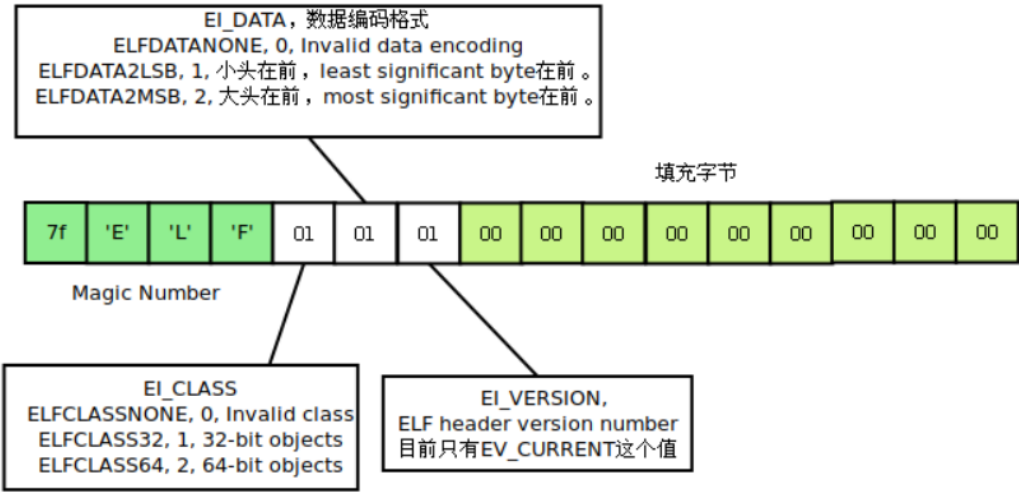


图 2-4 魔数具体分析

因此，Class 部分就是判断第 5 个字节，0 表示 Invalid class, 1 表示 32-bit objects, 2 表示 64-bit objects。Data 部分就是判断第 6 个字节，0 表示 Invalid data encoding, 1 表示 little endian, 2 表示 big endian。一般来说，文件的 Version、OS/ABI 以及 ABI Version 都是固定不变的。这些由于时间原因我没有深究，只做了简单的输出表达。

分析完了前 16 个字节，继续取 2 个字节，即得到 `e_type`，可以根据 `e_type` 来判断目标文件类型，1 表示可重定位文件，2 表示可执行文件，3 表示共享目标文件。其对应输出中的 Type 字段。

接着继续取 2 个字节，得到 `e_machine`，可根据 `e_machine` 来判断文件所处的硬件平台/机器类别。其中，2 表示 SPARC 机器，3 表示 386 机器，8 表示 MIPS 机器。具体可以根据 `elf.h` 信息判断。因为有很多，就只涉及了平常出现的部分。如图 2-5 所示。其对应输出中的 Machine 字段。

```

#define EM_NONE      0  /* No machine */
#define EM_M32       1  /* AT&T WE 32100 */
#define EM_SPARC     2  /* SUN SPARC */
#define EM_386       3  /* Intel 80386 */
#define EM_68K       4  /* Motorola m68k family */
#define EM_88K       5  /* Motorola m88k family */
#define EM_IAMCU     6  /* Intel MCU */
#define EM_860       7  /* Intel 80860 */
#define EM_MIPS      8  /* MIPS R3000 big-endian */
#define EM_S370      9  /* IBM System/370 */
#define EM_MIPS_RS3_LE 10 /* MIPS R3000 little-endian */
/* reserved 11-14 */
#define EM_PARISC    15 /* HPPA */
/* reserved 16 */
#define EM_VPP500    17 /* Fujitsu VPP500 */
#define EM_SPARC32PLUS 18 /* Sun's "v8plus" */
#define EM_960       19 /* Intel 80960 */
#define EM_PPC       20 /* PowerPC */
#define EM_PPC64     21 /* PowerPC 64-bit */
#define EM_S390      22 /* IBM S390 */
#define EM_SPU       23 /* IBM SPU/SPC */
/* reserved 24-35 */
#define EM_V800      36 /* NEC V800 series */
#define EM_FR20      37 /* Fujitsu FR20 */

```

图 2-5 部分机器类别

接着便到了 e_version 部分，取 4 个字节，进行输出即可。其中，1 表示当前版本。其对应输出中的 version 字段。接下来继续取 8 个字节得到 e_entry，因为是 64 位文件，所以要取 8 个字节（32 位要取 4 个字节），得到 e_entry，e_entry 给出程序入口的虚拟地址，即程序的开始位置。其对应输出中的 Entry point address 字段。接下来继续取 8 个字节（32 位取 4 个字节）得到 e_phoff，e_phoff 指出 program header table 的文件偏移量。其对应的是输出中的 Start of program headers 字段。接着继续取 8 个字节（32 位取 4 个字节）得到 e_shoff，e_shoff 指出节区头部表格的文件偏移量。其对应输出中的 Start of section headers 字段。接着继续取 4 个字节，得到 e_flags，其对应输出中的 Flags 字段。其表示处理器的特定标志，具体的我没有继续深究是什么特定标志。

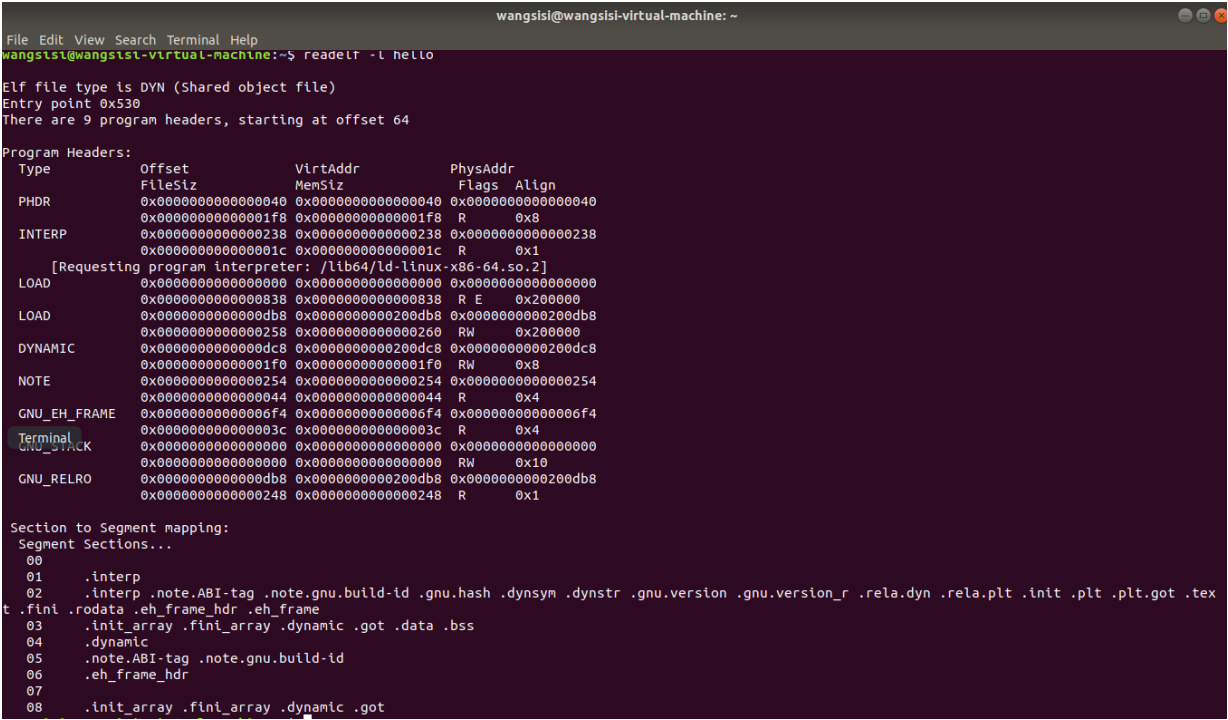
然后，取完 e_flags 之后继续向后面取 2 个字节，得到 e_ehsize，表示 ELF 头部的长度。对应的是输出中的 Size of this header 字段。然后，继续取 2 个字节，得到 e_phentsize，表示一个 program header table 中的条目的长度（字节数表示），其对应的是输出字段中的 Size of programme headers。然后继续取两位字节，得到 e_phnum，表示 program header table 中的条目数目，其对应的是输出语句中的 Number of section headers 字段。然后继续取两个字节得到 e_shentsize，表示 section header table 中的条目长度，继续取两位得到 e_shnum，表示 section header table 中的条目个数。其分别对应输出语句中的 Size of section headers 和 Number of section headers 字

段。最后，取两个字节，得到 e_shstrndx，其表示节区头部表格中与节区字符串表相关的表项索引。其对应的是输出语句中的 Section header string table index 字段。

至此，ELF 文件头解析完毕。

2.2 解析程序头表

首先使用 readelf 查看程序头表中的信息，如图 2-6 所示。



```
wanglisi@wanglisi-virtual-machine: ~  
File Edit View Search Terminal Help  
wanglisi@wanglisi-virtual-machine:~$ readelf -l hello  
  
Elf file type is DYN (Shared object file)  
Entry point 0x530  
There are 9 program headers, starting at offset 64  
  
Program Headers:  
Type           Offset             VirtAddr           PhysAddr  
FileSiz        MemSiz             Flags             Align  
PHDR            0x0000000000000040 0x0000000000000040 0x0000000000000040  
0x00000000000001f8 0x00000000000001f8 R               0x8  
INTERP          0x0000000000000238 0x0000000000000238 0x0000000000000238  
0x000000000000001c 0x000000000000001c R               0x1  
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]  
LOAD            0x0000000000000000 0x0000000000000000 0x0000000000000000  
0x0000000000000038 0x0000000000000038 R E             0x200000  
0x0000000000000db8 0x00000000000020db8 0x00000000000020db8  
0x000000000000258 0x000000000000258 RW             0x200000  
DYNAMIC         0x0000000000000dc8 0x00000000000020dc8 0x00000000000020dc8  
0x00000000000001f0 0x00000000000001f0 RW             0x8  
0x000000000000254 0x000000000000254 0x000000000000254  
0x000000000000044 0x000000000000044 R               0x4  
GNU_EH_FRAME    0x00000000000006f4 0x00000000000006f4 0x00000000000006f4  
0x000000000000003c 0x000000000000003c R               0x4  
GNU_STACK       0x0000000000000000 0x0000000000000000 0x0000000000000000  
0x0000000000000000 0x0000000000000000 RW             0x10  
GNU_RELRO       0x0000000000000db8 0x00000000000020db8 0x00000000000020db8  
0x000000000000248 0x000000000000248 R               0x1  
  
Section to Segment mapping:  
Segment Sections...  
00  
01 .interp  
02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame  
03 .init_array .fini_array .dynamic .got .data .bss  
04 .dynamic  
05 .note.ABI-tag .note.gnu.build-id  
06 .eh_frame_hdr  
07  
08 .init_array .fini_array .dynamic .got
```

图 2-6 程序头表查询

下面图 2-7 是我用 C 语言解析文件得出的程序头表信息。

```
E:\linux\大作业\新建文件夹\2_64.exe
Elf file type is Shared object file
Entry point 0X530
There are 9 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz          MemSiz              Flags      Align
PHDR           0000000000000040   0000000000000040   0000000000000040
               00000000000001F8   00000000000001F8   R          8
INTERP         0000000000000238   0000000000000238   0000000000000238
               000000000000001C   000000000000001C   R          1
               [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0000000000000000   0000000000000000   0000000000000000
               0000000000000838   0000000000000838   R E        200000
LOAD           0000000000000DB8   0000000000200DB8   0000000000200DB8
               0000000000000258   0000000000000260   RW         200000
DYNAMIC         0000000000000DC8   0000000000200DC8   0000000000200DC8
               00000000000001F0   00000000000001F0   RW          8
NOTE           0000000000000254   0000000000000254   0000000000000254
               0000000000000044   0000000000000044   R          4
GNU_EH_FRAME    00000000000006F4   00000000000006F4   00000000000006F4
               000000000000003C   000000000000003C   R          4
GNU_STACK       0000000000000000   0000000000000000   0000000000000000
               0000000000000000   0000000000000000   RW         10
GNU_RELRO       0000000000000DB8   0000000000200DB8   0000000000200DB8
               0000000000000248   0000000000000248   R          1

Section to Segment mapping:
Segment Sections...
0      .strtab .shstrtab
1      .interp
2      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela .dyn .rela
.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame .comment .symtab .strtab .shstrtab
3      .init_array .fini_array .dynamic .got .data .bss
4      .dynamic
5      .note.ABI-tag .note.gnu.build-id
6      .eh_frame_hdr
7
8      .init_array .fini_array .dynamic .got

-----
Process exited after 0.6006 seconds with return value 0
请按任意键继续. . .

微软拼音 半:
```

图 2-7 C 语言查询程序头表

可执行文件或者共享目标文件紧接着 ELF 头部的就是程序头表，程序头表是一个结构数组，包含了 ELF 头表中字段 `e_phnum` 个定义的条目。每个结构描述了一个段或者系统准备程序执行所必需的其他信息。目标文件中的“段”包括一个或者多个节区，也就是“段内容 (Segment Contents)”。程序头部仅对可执行文件和共享目标文件有意义。根据 ELF 头部的 `e_phsize` 和 `e_phnum` 可以得出程序头部的大小。

对于分析 ELF 文件的程序头表，首先定义一个 ELF 头文件结构的指针，将其指向 ELF 文件的头部，这样就可以得到 ELF 头部文件中与程序头表相关的信息。根据 `e_type` 得出 ELF 文件类型，根据 `e_entry` 得出程序进入点，即 Entry point。然后再根据 `e_phnum` 和 `e_hoff` 得出程序头的个数以及其偏移量。

创建一个程序头表结构指针 `proheader`，然后指向 EFL 头文件指针加上 `e_phoff`（偏移量），就是程序头表所处位置。根据 ELF 头文件指针得到的 `e_phnum` 和 `proheader` 指针来遍历程序头表。在输出的时候，根据 `p_type` 来区分不同种类的段（如图 2-8 所示）并输出。当遇到 `PT_INTERP` 时，表明了运行此程序所需要的程序解释器（`/lib/ls-linux.so.2`）。

```

#define PT_NULL      0      /* Program header table entry unused */
#define PT_LOAD      1      /* Loadable program segment */
#define PT_DYNAMIC    2      /* Dynamic linking information */
#define PT_INTERP     3      /* Program interpreter */
#define PT_NOTE       4      /* Auxiliary information */
#define PT_SHLIB      5      /* Reserved */
#define PT_PHDR       6      /* Entry for header table itself */
#define PT_TLS        7      /* Thread-local storage segment */
#define PT_NUM        8      /* Number of defined types */
#define PT_LOOS       0x60000000 /* Start of OS-specific */
#define PT_GNU_EH_FRAME 0x6474e550 /* GCC .eh_frame_hdr segment */
#define PT_GNU_STACK  0x6474e551 /* Indicates stack executability */
#define PT_GNU_RELRO   0x6474e552 /* Read-only after relocation */
#define PT_LOSUNW     0x6ffffffa
#define PT_SUNWBSS     0x6ffffffa /* Sun Specific segment */
#define PT_SUNWSTACK   0x6ffffffb /* Stack segment */
#define PT_HISUNW     0x6fffffff
#define PT_HIOS        0x6fffffff /* End of OS-specific */
#define PT_LOPROC      0x70000000 /* Start of processor-specific */
#define PT_HIPROC      0x7fffffff /* End of processor-specific */

```

图 2-8 段的不同类型

根据 `p_flags` 不同来输出段的不同标记（如图 2-9 所示）。

```

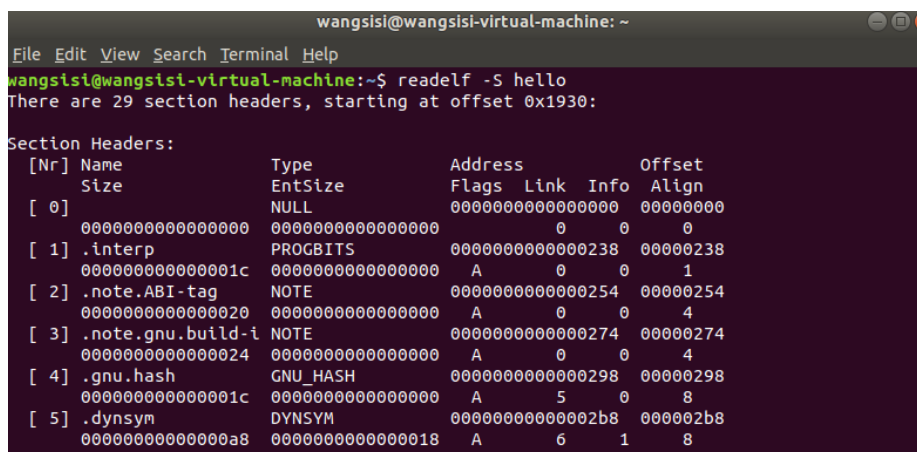
/* Legal values for p_flags (segment flags). */
#define PF_X          (1 << 0) /* Segment is executable */
#define PF_W          (1 << 1) /* Segment is writable */
#define PF_R          (1 << 2) /* Segment is readable */
#define PF_MASKOS     0x0ff00000 /* OS-specific */
#define PF_MASKPROC    0xf0000000 /* Processor-specific */

```

图 2-9 段的不同标记

以上，就可以输出程序头里面的所有段信息了。

对于后面的 Section to Segment mapping 一开始，我是看不出来它是怎么计算的，然后经过查阅资料和自己的验证，才得出它的产生方法。首先用 `readelf` 文件查看，得出节头表如图 2-10 所示。



```

wangsisi@wangsisi-virtual-machine: ~
File Edit View Search Terminal Help
wangsisi@wangsisi-virtual-machine:~$ readelf -S hello
There are 29 section headers, starting at offset 0x1930:

Section Headers:
 [Nr] Name              Type              Address            Offset
     Size              EntSize          Flags Link Info  Align
 [ 0]                      NULL              0000000000000000  00000000
     0000000000000000  0000000000000000  0 0 0
 [ 1] .interp             PROGBITS          0000000000000238  00000238
     000000000000001c  0000000000000000  A 0 0 1
 [ 2] .note.ABI-tag       NOTE              0000000000000254  00000254
     0000000000000020  0000000000000000  A 0 0 4
 [ 3] .note.gnu.build-id  NOTE              0000000000000274  00000274
     0000000000000024  0000000000000000  A 0 0 4
 [ 4] .gnu.hash           GNU_HASH          0000000000000298  00000298
     000000000000001c  0000000000000000  A 5 0 8
 [ 5] .dynsym             DYNSYM            00000000000002b8  000002b8
     00000000000000a8  0000000000000018  A 6 1 8

```

图 2-10 节头表部分信息

对于图 2-11 中的 01 号字段分析。他的首地址段位 0x0000000000000238, 大小位 0x000000000000001c。刚好对应图 2-12 中的 .interp 节的地址和大小。

```
File Edit View Search Terminal Help
Entry point 0x530
There are 9 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz          MemSiz          Flags  Align
PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
               0x00000000000001f8 0x00000000000001f8 R      0x8
INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
               0x000000000000001c 0x000000000000001c R      0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD          0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000838 0x0000000000000838 R E    0x200000
LOAD          0x0000000000000db8 0x0000000000020db8 0x0000000000020db8
               0x000000000000258 0x000000000000260 RW     0x200000
DYNAMIC        0x000000000000dc8 0x0000000000020dc8 0x0000000000020dc8
               0x0000000000001f0 0x0000000000001f0 RW     0x8
NOTE          0x000000000000254 0x000000000000254 0x000000000000254
               0x000000000000044 0x000000000000044 R      0x4
GNU_EH_FRAME   0x00000000000006f4 0x00000000000006f4 0x00000000000006f4
               0x00000000000003c 0x00000000000003c R      0x4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000 RW     0x10
GNU_RELRO      0x000000000000db8 0x0000000000020db8 0x0000000000020db8
               0x000000000000248 0x000000000000248 R      0x1

Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu
u.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .ro
data .eh_frame_hdr .eh_frame
03 .init_array .fini_array .dynamic .got .data .bss
```

图 2-11

```
wangsysi@wangsysi-virtual-machine: ~
File Edit View Search Terminal Help
wangsysi@wangsysi-virtual-machine:~$ readelf -S hello
There are 29 section headers, starting at offset 0x1930:

Section Headers:
[Nr] Name           Type              Address            Offset
     Size            EntSize          Flags    Link    Info    Align
[ 0]                  NULL              0000000000000000   00000000
     0000000000000000 0000000000000000   0          0      0
[ 1] .interp           PROGBITS          0000000000000238   00000238
     000000000000001c 0000000000000000   A          0      1
[ 2] .note.ABI-tag     NOTE              0000000000000254   00000254
     0000000000000020 0000000000000000   A          0      4
[ 3] .note.gnu.build-i NOTE              0000000000000274   00000274
     0000000000000024 0000000000000000   A          0      4
[ 4] .gnu.hash         GNU_HASH          0000000000000298   00000298
     000000000000001c 0000000000000000   A          5      8
[ 5] .dynsym           DYNAMIC           00000000000002b8   000002b8
     0000000000000a8 000000000000018   A          6      1
```

图 2-12

然后再继续对 03 段分析，观察如图 2-11 段的初始位置和段的大小，再去观察图 2-13 中段对应的小节部分，两个初始位置相同，各个小节大小总和等于段的大小。段就是由多个小节组成的。

```
0000000000000108 0000000000000000 A 0 0 8
[19] .init_array      INIT_ARRAY        0000000000020db8 00000db8
     0000000000000008 0000000000000008 WA 0 0 8
[20] .fini_array      FINI_ARRAY        0000000000020dc0 00000dc0
     0000000000000008 0000000000000008 WA 0 0 8
[21] .dynamic         DYNAMIC           0000000000020dc8 00000dc8
     00000000000001f0 0000000000000010 WA 6 0 8
[22] .got             PROGBITS          0000000000020fb8 00000fb8
     0000000000000048 0000000000000008 WA 0 0 8
[23] .data            PROGBITS          0000000000020100 00001000
     0000000000000010 0000000000000000 WA 0 0 8
[24] .bss             NOBITS            0000000000020100 00001010
     0000000000000008 0000000000000000 WA 0 0 1
```

图 2-13

知道了 progame header table 和 section header table 之间的关系，就可以得出输出字段的 Section to Segment mapping 相应信息了。定义一个 section header table 对应的指针，指向 section header table 中，根据段的 p_vaddr 和 p_memsz 以及 section header table 中的 sh_addr 和 sh_size 进行判断，最后得出 Section to Segment mapping。

至此，程序头表分析完毕。

2.3 解析节头表

首先使用 readelf 工具查看节头表中的信息。如图 2-14，图 2-15 所示。

```
wanglisi@wanglisi-virtual-machine: ~  
File Edit View Search Terminal Help  
wanglisi@wanglisi-virtual-machine:~$ readelf -S hello  
There are 29 section headers, starting at offset 0x1930:  
  
Section Headers:  
[Nr] Name                Type              Address            Offset  
Size              EntSize          Flags Link Info Align  
[ 0] 0000000000000000 NULL              0000000000000000 00000000  
[ 1] .interp               PROGBITS          000000000000238 0000238  
000000000000001c 0000000000000000 A 0 0 1  
[ 2] .note.ABI-tag         NOTE              000000000000254 0000254  
0000000000000020 0000000000000000 A 0 0 4  
[ 3] .note.gnu.build-id    NOTE              000000000000274 0000274  
0000000000000024 0000000000000000 A 0 0 4  
[ 4] .gnu.hash             GNU_HASH          000000000000298 0000298  
000000000000001c 0000000000000000 A 5 0 8  
[ 5] .dynsym               DYNSYM            0000000000002b8 00002b8  
00000000000000a8 0000000000000018 A 6 1 8  
[ 6] .dynstr               STRTAB            000000000000360 0000360  
0000000000000082 0000000000000000 A 0 0 1  
[ 7] .gnu.version          VERSYM            0000000000003e2 00003e2  
000000000000000e 0000000000000002 A 5 0 2  
[ 8] .gnu.version_r        VERNEED           0000000000003f0 00003f0  
0000000000000020 0000000000000000 A 6 1 8  
[ 9] .rela.dyn             RELA              000000000000410 0000410  
00000000000000c0 0000000000000018 A 5 0 8  
[10] .rela.plt             RELA              0000000000004d0 00004d0  
0000000000000018 0000000000000018 AI 5 22 8  
[11] .init                 PROGBITS          0000000000004e8 00004e8  
0000000000000017 0000000000000000 AX 0 0 4  
[12] .plt                 PROGBITS          000000000000500 0000500  
0000000000000070 0000000000000010 AX 0 0 16
```

图 2-14 节头表 1

```
wanglisi@wanglisi-virtual-machine: ~  
File Edit View Search Terminal Help  
[13] .plt.got              PROGBITS          000000000000520 0000520  
0000000000000008 0000000000000008 AX 0 0 8  
[14] .text                PROGBITS          000000000000530 0000530  
00000000000001a2 0000000000000000 AX 0 0 16  
[15] .fini                PROGBITS          0000000000006d4 00006d4  
0000000000000009 0000000000000000 AX 0 0 4  
[16] .rodata              PROGBITS          0000000000006e0 00006e0  
0000000000000011 0000000000000000 A 0 0 4  
[17] .eh_frame_hdr        PROGBITS          0000000000006f4 00006f4  
000000000000003c 0000000000000000 A 0 0 4  
[18] .eh_frame            PROGBITS          000000000000730 0000730  
0000000000000108 0000000000000000 A 0 0 8  
[19] .init_array           INIT_ARRAY        000000000020db8 0000db8  
0000000000000008 0000000000000008 WA 0 0 8  
[20] .fini_array           FINI_ARRAY        000000000020dc0 0000dc0  
0000000000000008 0000000000000008 WA 0 0 8  
[21] .dynamic              DYNAMIC           000000000020dc8 0000dc8  
00000000000001f0 0000000000000010 WA 6 0 8  
[22] .got                 PROGBITS          000000000020fb8 0000fb8  
0000000000000048 0000000000000008 WA 0 0 8  
[23] .data                PROGBITS          000000000021000 0001000  
0000000000000010 0000000000000000 WA 0 0 8  
[24] .bss                 NOBITS            000000000021010 0001010  
0000000000000008 0000000000000000 WA 0 0 1  
[25] .comment              PROGBITS          0000000000001010 0001010  
0000000000000029 0000000000000001 M5 0 0 1  
[26] .syntab              SYMTAB            0000000000001040 0001040  
00000000000005e8 0000000000000018 27 43 8  
[27] .strtab              STRTAB            0000000000001628 0001628  
0000000000000203 0000000000000000 0 0 1  
[28] .shstrtab            STRTAB            000000000000182b 000182b  
00000000000000fe 0000000000000000 0 0 1  
  
Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
L (link order), O (extra OS processing required), G (group), T (TLS),  
C (compressed), x (unknown), o (OS specific), E (exclude),  
l (large), p (processor specific)  
wanglisi@wanglisi-virtual-machine:~$
```

图 2-14 节头表 2

图 2-15，图 2-16 是 C 语言得出的结果。

```
E:\linux\大作业\新建文件夹\3_64.exe
There are 29 section headers, starting at offset 0x1930:

Section Headers:
[Nr] Name              Type              Address            Offset
    Size              EntSize          Flags Link    Info  Align
[ 0] 0000000000000000 NULL              0000000000000000 0 0 0
[ 1] .interp              PROGBITS          0000000000000238 00000238
    000000000000001C 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag        NOTE              0000000000000254 00000254
    0000000000000020 0000000000000000 A 0 0 4
[ 3] .note.gnu.build-id   NOTE              0000000000000274 00000274
    0000000000000024 0000000000000000 A 0 0 4
[ 4] .gnu.hash            GNU_HASH          0000000000000298 00000298
    000000000000001C 0000000000000000 A 5 0 8
[ 5] .dynsym              DYNSYM            00000000000002B8 000002B8
    00000000000000A8 0000000000000018 A 6 1 8
[ 6] .dynstr              STRTAB            0000000000000360 00000360
    0000000000000082 0000000000000000 A 0 0 1
[ 7] .gnu.version          GNU_verSYM        00000000000003E2 000003E2
    000000000000000E 0000000000000002 A 5 0 2
[ 8] .gnu.version_r        GNU_verneed       00000000000003F0 000003F0
    0000000000000020 0000000000000000 A 6 1 8
[ 9] .rela.dyn             RELA              0000000000000410 00000410
    00000000000000C0 0000000000000018 A 5 0 8
[10] .rela.plt             RELA              00000000000004D0 000004D0
    0000000000000018 0000000000000018 NONE
[11] .init                 PROGBITS          00000000000004E8 000004E8
    0000000000000017 0000000000000000 AX 0 0 4
[12] .plt                  PROGBITS          0000000000000500 00000500
    0000000000000020 0000000000000010 AX 0 0 16
[13] .plt.got              PROGBITS          0000000000000520 00000520
    0000000000000008 0000000000000008 AX 0 0 8
```

图 2-15 C 解析节头表

```
E:\linux\大作业\新建文件夹\3_64.exe
[13] .plt.got              PROGBITS          0000000000000520 00000520
    0000000000000008 0000000000000008 AX 0 0 8
[14] .text                 PROGBITS          0000000000000530 00000530
    00000000000001A2 0000000000000000 AX 0 0 16
[15] .fini                 PROGBITS          00000000000006D4 000006D4
    0000000000000009 0000000000000000 AX 0 0 4
[16] .rodata               PROGBITS          00000000000006E0 000006E0
    0000000000000011 0000000000000000 A 0 0 4
[17] .eh_frame_hdr         PROGBITS          00000000000006F4 000006F4
    000000000000003C 0000000000000000 A 0 0 4
[18] .eh_frame             PROGBITS          0000000000000730 00000730
    0000000000000108 0000000000000000 A 0 0 8
[19] .init_array           INIT_ARRAY        0000000000000DB8 00000DB8
    0000000000000008 0000000000000008 WA 0 0 8
[20] .fini_array           FINI_ARRAY        0000000000000DC0 00000DC0
    0000000000000008 0000000000000008 WA 0 0 8
[21] .dynamic               DYNAMIC           0000000000000DC8 00000DC8
    00000000000001F0 0000000000000010 WA 6 0 8
[22] .got                  PROGBITS          0000000000000FB8 00000FB8
    0000000000000048 0000000000000008 WA 0 0 8
[23] .data                 PROGBITS          0000000000001000 00001000
    0000000000000010 0000000000000000 WA 0 0 8
[24] .bss                  NOBITS            0000000000001010 00001010
    0000000000000008 0000000000000000 WA 0 0 1
[25] .comment              PROGBITS          0000000000001010 00001010
    0000000000000029 0000000000000001 NONE
[26] .symtab               SYMTAB            0000000000001040 00001040
    00000000000005E8 0000000000000018 27 43 8
[27] .strtab               STRTAB            0000000000000000 00001628
    0000000000000203 0000000000000000 0 0 1
[28] .shstrtab             STRTAB            0000000000000000 0000182B
    00000000000000FE 0000000000000000 0 0 1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (large)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

-----
Process exited after 0.6172 seconds with return value 0
请按任意键继续. . .
```

图 2-16 C 解析节头表

解析 ELF 文件中的节头表，与解析程序头表相似，首先定义一个 ELF 头文件指针指向 ELF 的头文件，得出 `e_shnum` 和 `e_sthstrndx`，从而得到节头表的大小以及相应的字符索引。然后定义一个节头表结构指针，根据 `e_shoff` 指向 ELF 文件的节头表位置。

根据节头表的大小进行遍历输出节头表即可。因为 `sh_name` 代表的是小节名在字符串表中的索引，所以，还需要另外的一个指针去定位到字符串表，然

后才能输出每个 section 的名称。Elf 文件中用到的字符串，如段名、函数名、变量名称等，均保存在字符串表中。其中，shstrtab 段表字符串表仅用来保存段名，而 strtab 或 dynstr section 则是存放普通字符串，如函数、变量名等符号名称，字符串之间以“00”截断。

其中，首先读出 ELF 文件的头文件中的 e_shstrndx，然后创建一个 section 结构的指针，指向文件节头表加上 e_shstrndx 的地方，这就是符号表的偏移位置，然后就可以使用一个字符指针去指向字符串表。所以，在遍历节头表时，就能够输出 sh_name 所对应的 section 的名称了。

在遍历数组时，每一个 section header 都有相应的 sh_type，根据不同类型进行分别输出，如图 2-17 所示。

```
#define SHT_NULL 0 /* Section header table entry unused */
#define SHT_PROGBITS 1 /* Program data */
#define SHT_SYMTAB 2 /* Symbol table */
#define SHT_STRTAB 3 /* String table */
#define SHT_RELA 4 /* Relocation entries with addends */
#define SHT_HASH 5 /* Symbol hash table */
#define SHT_DYNAMIC 6 /* Dynamic linking information */
#define SHT_NOTE 7 /* Notes */
#define SHT_NOBITS 8 /* Program space with no data (bss) */
#define SHT_REL 9 /* Relocation entries, no addends */
#define SHT_SHLIB 10 /* Reserved */
#define SHT_DYNSYM 11 /* Dynamic linker symbol table */
#define SHT_INIT_ARRAY 14 /* Array of constructors */
#define SHT_FINI_ARRAY 15 /* Array of destructors */
#define SHT_PREINIT_ARRAY 16 /* Array of pre-constructors */
#define SHT_GROUP 17 /* Section group */
#define SHT_SYMTAB_SHNDX 18 /* Extended section indexes */
#define SHT_NUM 19 /* Number of defined types. */
#define SHT_LOOS 0x60000000 /* Start OS-specific. */
#define SHT_GNU_ATTRIBUTES 0x6ffffff5 /* Object attributes. */
#define SHT_GNU_HASH 0x6ffffff6 /* GNU-style hash table. */
#define SHT_GNU_LIBLIST 0x6ffffff7 /* Prelink library list */
#define SHT_CHECKSUM 0x6ffffff8 /* Checksum for DSO content. */
#define SHT_LOSUNW 0x6ffffffa /* Sun-specific low bound. */
```

图 2-17 部分 sh_type

最后依次输出 section header 即可。在末尾再加上相应的说明语句。

至此，节头表解析完毕。

2.4 解析符号表

在链接过程中，函数和变量统称为符号，函数名或变量名就是符号名。符号表的段名为 symtab 或 dynsym，它是一个 Elf32/64_Sym 结构的数组。

readelf 工具可以查看 ELF 文件的符号表中的 Symbol Table，以及其中包含的具体内容。所以便对符号表进行深入研究。

首先 readelf 工具产生的符号表如图 2-18 所示。

```
wangsis@wangsis-virtual-machine: ~
File Edit View Search Terminal Help
wangsis@wangsis-virtual-machine:~$ readelf -s 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_deregisterTMCloneTable
    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 63 entries:
  Num:  Value              Size Type Bind Vis      Ndx Name
    0:  0000000000000000      0 NOTYPE LOCAL DEFAULT UND
    1:  0000000000000238      0 SECTION LOCAL DEFAULT 1
    2:  0000000000000254      0 SECTION LOCAL DEFAULT 2
    3:  0000000000000274      0 SECTION LOCAL DEFAULT 3
```

图 2-18 readelf 产生符号表

使用 C 语言所产生的符号表，如图 2-19 所示。

```
E:\linux\大作业\新建文件夹\4_64.exe
Symbol table '.dynsym' contains 7 entries:
  Num:  Value              Size Type Bind Vis      Ndx Name
    0:  0000000000000000      0 NOTYPE LOCAL 0 UND
    1:  0000000000000000      0 NOTYPE WEAK 0 UND _ITM_deregisterTMCloneTable
    2:  0000000000000000      0 FUNC  GLOBAL 0 UND puts
    3:  0000000000000000      0 FUNC  GLOBAL 0 UND __libc_start_main
    4:  0000000000000000      0 NOTYPE WEAK 0 UND __gmon_start__
    5:  0000000000000000      0 NOTYPE WEAK 0 UND _ITM_registerTMCloneTable
    6:  0000000000000000      0 FUNC  WEAK 0 UND __cxa_finalize

Symbol table '.symtab' contains 63 entries:
  Num:  Value              Size Type Bind Vis      Ndx Name
    0:  0000000000000000      0 NOTYPE LOCAL 0 UND
    1:  0000000000000238      0 SECTION LOCAL 0 1
    2:  0000000000000254      0 SECTION LOCAL 0 2
    3:  0000000000000274      0 SECTION LOCAL 0 3
    4:  0000000000000298      0 SECTION LOCAL 0 4
    5:  00000000000002b8      0 SECTION LOCAL 0 5
    6:  0000000000000360      0 SECTION LOCAL 0 6
    7:  00000000000003e2      0 SECTION LOCAL 0 7
    8:  00000000000003f0      0 SECTION LOCAL 0 8
    9:  0000000000000410      0 SECTION LOCAL 0 9
   10:  00000000000004d0      0 SECTION LOCAL 0 10
   11:  00000000000004e8      0 SECTION LOCAL 0 11
   12:  0000000000000500      0 SECTION LOCAL 0 12
   13:  0000000000000520      0 SECTION LOCAL 0 13
   14:  0000000000000530      0 SECTION LOCAL 0 14
   15:  00000000000006d4      0 SECTION LOCAL 0 15
   16:  00000000000006e0      0 SECTION LOCAL 0 16
   17:  00000000000006f4      0 SECTION LOCAL 0 17
微软拼音 半:
```

图 2-19 C 语言产生符号表

目标文件中的“符号表(symbol table)”所包含的信息用于定位和重定位程序中的符号定义和引用。目标文件的其它部分通过一个符号在这个表中的索引值来使用该符号。索引值从 0 开始计数,但值为 0 的表项(即第一项)并没有实际的意义,它表示未定义的符号。它的数据结构如图 2-20, 图 2-21 所示。

```
wanglisi@wanglisi-virtual-machine: ~
File Edit View Search Terminal Help
typedef struct
{
    Elf32_Word    st_name;           /* Symbol name (string tbl index) */
    Elf32_Addr    st_value;          /* Symbol value */
    Elf32_Word    st_size;           /* Symbol size */
    unsigned char st_info;           /* Symbol type and binding */
    unsigned char st_other;          /* Symbol visibility */
    Elf32_Section st_shndx;          /* Section index */
} Elf32_Sym;
```

图 2-20 符号表项结构（32 位）

```
wanglisi@wanglisi-virtual-machine: ~
File Edit View Search Terminal Help
typedef struct
{
    Elf64_Word    st_name;           /* Symbol name (string tbl index) */
    unsigned char st_info;           /* Symbol type and binding */
    unsigned char st_other;          /* Symbol visibility */
    Elf64_Section st_shndx;          /* Section index */
    Elf64_Addr    st_value;          /* Symbol value */
    Elf64_Xword   st_size;           /* Symbol size */
} Elf64_Sym;
```

图 2-21 符号表项结构（64 位）

st_name	符号的名字，指向字符串表的索引值
st_value	符号的值
st_size	符号的大小
st_info	符号的类型和属性
st_other	其他信息
st_shndx	相关与节的索引

表 4 表项的具体含义

解析符号表，首先定义 section header 的数据结构，指向 ELF 文件的节头表，找出 .symtab/和 .dynsym 节，然后根据 sh_offset 来找到符号表所处位置，并创建一个符号表结构指针指向该位置。根据 sh_entsize 来得出符号表中的表项数目。这样就可以遍历出符号表中的每一个表项了。

其中，每个表项的 st_info 是不同的，st_info 由一系列的比特位构成，标识了“符号绑定(symbol binding)”、“符号类型(symbol type)”和“符号信息(symbol infomation)”三种属性。下面几个宏分别用于读取这三种属性值。

```
#define ELF32/64_ST_BIND(i) ((i)>>4)
#define ELF32/64_ST_TYPE(i) ((i)&0xf)
#define ELF32/64_ST_INFO(b, t) (((b)<<4)+((t)&0xf))
```

符号绑定 (Symbol Binding):符号绑定属性由 ELF32/64_ST_BIND 指定。如图 2-22 所示。


```

/* Legal values for ST_BIND subfield of st_info (symbol binding). */

#define STB_LOCAL 0 /* Local symbol */
#define STB_GLOBAL 1 /* Global symbol */
#define STB_WEAK 2 /* Weak symbol */
#define STB_NUM 3 /* Number of defined types. */
#define STB_LOOS 10 /* Start of OS-specific */
#define STB_GNU_UNIQUE 10 /* Unique symbol. */
#define STB_HIOS 12 /* End of OS-specific */
#define STB_LOPROC 13 /* Start of processor-specific */
#define STB_HIPROC 15 /* End of processor-specific */

```

图 2-22 符号绑定

符号类型 (Symbol Types) : 符号类型属性由 ELF32/64_ST_TYPE 指定。如图 2-23 所示。

```

/* Legal values for ST_TYPE subfield of st_info (symbol type). */

#define STT_NOTYPE 0 /* Symbol type is unspecified */
#define STT_OBJECT 1 /* Symbol is a data object */
#define STT_FUNC 2 /* Symbol is a code object */
#define STT_SECTION 3 /* Symbol associated with a section */
#define STT_FILE 4 /* Symbol's name is file name */
#define STT_COMMON 5 /* Symbol is a common data object */
#define STT_TLS 6 /* Symbol is thread-local data object */
#define STT_NUM 7 /* Number of defined types. */
#define STT_LOOS 10 /* Start of OS-specific */
#define STT_GNU_IFUNC 10 /* Symbol is indirect code object */
#define STT_HIOS 12 /* End of OS-specific */
#define STT_LOPROC 13 /* Start of processor-specific */
#define STT_HIPROC 15 /* End of processor-specific */

```

图 2-23 符号类型

不同的表项中的 st_shndx 也不同, 任何一个符号表项的定义都与某一个“节”相联系, 因为符号是为节而定义, 在节中被引用。本数据成员即指明了相关联的节。本数据成员是一个索引值, 它指向相关联的节在节头表中的索引。在重定位过程中, 节的位置会改变, 本数据成员的值也随之改变, 继续指向节的新位置。具体如图 2-24 所示。

```

/* Special section indices. */

#define SHN_UNDEF 0 /* Undefined section */
#define SHN_LORESERVE 0xff00 /* Start of reserved indices */
#define SHN_LOPROC 0xff00 /* Start of processor-specific */
#define SHN_BEFORE 0xff00 /* Order section before all others
(Solaris). */
#define SHN_AFTER 0xff01 /* Order section after all others
(Solaris). */
#define SHN_HIPROC 0xff1f /* End of processor-specific */
#define SHN_LOOS 0xff20 /* Start of OS-specific */
#define SHN_HIOS 0xff3f /* End of OS-specific */
#define SHN_ABS 0xffff1 /* Associated symbol is absolute */
#define SHN_COMMON 0xffff2 /* Associated symbol is common */
#define SHN_XINDEX 0xfffff /* Index is in extra table. */
#define SHN_HIRESERVE 0xfffff /* End of reserved indices */

```

表 2-24 st_shndx

其中, SHN_ABS: 符号的值是绝对的, 具有常量性, 在重定位过程中, 此值不需要改变。在输出语句中对应 ABS。

SHN_COMMON: 本符号所关联的是一个还没有分配的公共节, 本符号的值规定了其内容的字节对齐规则, 与 sh_addralign 相似。也就是说, 连接器会为本符号分配存储空间, 而且其起始地址是向 st_value 对齐的。本符号的值指明了要分配的字节数。在输出语句中对应 COM。

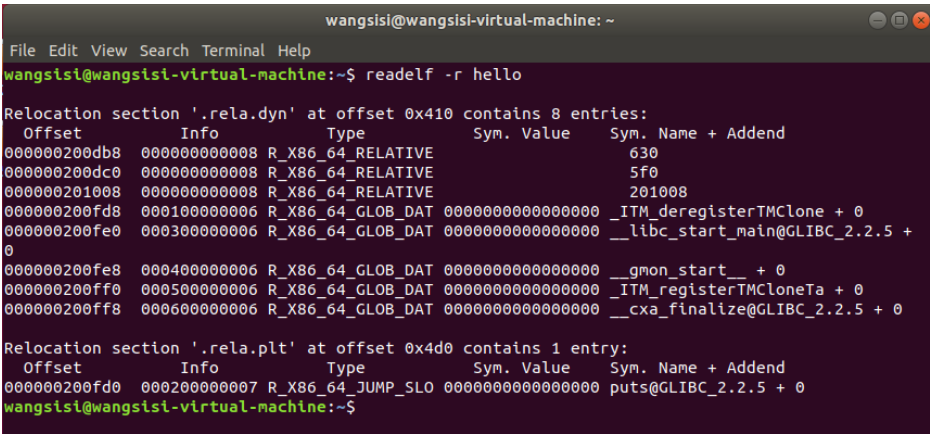
SHN_UNDEF: 表明本符号在当前目标文件中未定义, 在连接过程中, 连接器会找到此符号被定义的文件, 并把这些文件连接在一起。本文件对该符号的引用会被连接到实际的定义上去。在输出语句中对应 UND。

至此, 符号表解析完毕。

2.5 解析重定位表

重定位是将符号引用与符号定义进行链接的过程。例如, 当程序调用了一个函数时, 相关的调用指令必须把控制传输到适当的目标执行地址。

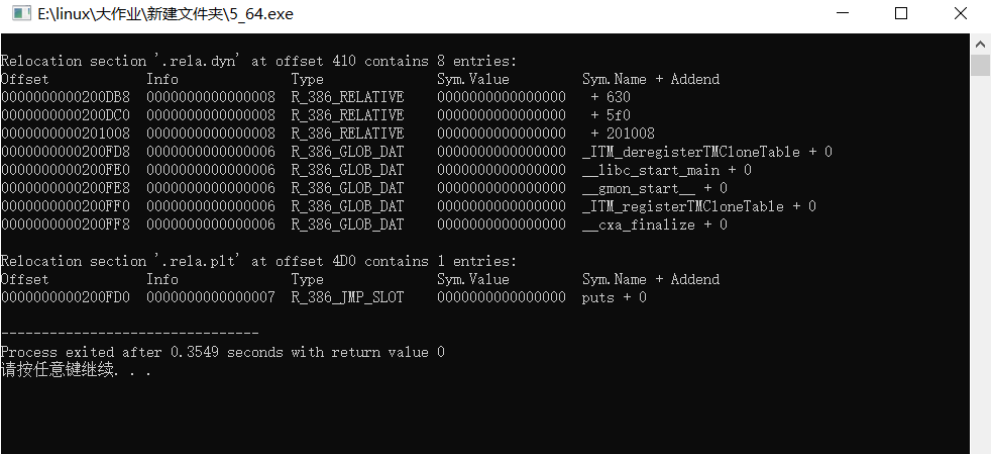
图 2-25 是, readelf 工具所产生的重定位表。



```
wangsisl@wangsisl-virtual-machine: ~  
File Edit View Search Terminal Help  
wangsisl@wangsisl-virtual-machine:~$ readelf -r hello  
  
Relocation section '.rela.dyn' at offset 0x410 contains 8 entries:  
  Offset      Info          Type           Sym. Value     Sym. Name + Addend  
000000200db8  0000000000000008 R_X86_64_RELATIVE 0000000000000000 + 630  
000000200dc0  0000000000000008 R_X86_64_RELATIVE 0000000000000000 + 5f0  
000000201008  0000000000000008 R_X86_64_RELATIVE 0000000000000000 + 201008  
000000200fd8  0001000000000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTMClone + 0  
000000200fe0  0003000000000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0  
000000200fe8  0004000000000006 R_X86_64_GLOB_DAT 0000000000000000 _gmon_start__ + 0  
000000200ff0  0005000000000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCloneTa + 0  
000000200ff8  0006000000000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0  
  
Relocation section '.rela.plt' at offset 0x4d0 contains 1 entry:  
  Offset      Info          Type           Sym. Value     Sym. Name + Addend  
000000200fd0  0002000000000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0  
wangsisl@wangsisl-virtual-machine:~$
```

图 2-25 readelf 产生重定位表

图 2-26 是 C 语言产生的重定位表。



```
E:\linux\大作业\新建文件夹\5_64.exe  
Relocation section '.rela.dyn' at offset 410 contains 8 entries:  
Offset      Info          Type           Sym. Value     Sym. Name + Addend  
0000000000200db8  0000000000000008 R_386_RELATIVE 0000000000000000 + 630  
0000000000200dc0  0000000000000008 R_386_RELATIVE 0000000000000000 + 5f0  
0000000000201008  0000000000000008 R_386_RELATIVE 0000000000000000 + 201008  
0000000000200fd8  0000000000000006 R_386_GLOB_DAT 0000000000000000 _ITM_deregisterTMCloneTable + 0  
0000000000200fe0  0000000000000006 R_386_GLOB_DAT 0000000000000000 __libc_start_main + 0  
0000000000200fe8  0000000000000006 R_386_GLOB_DAT 0000000000000000 _gmon_start__ + 0  
0000000000200ff0  0000000000000006 R_386_GLOB_DAT 0000000000000000 _ITM_registerTMCloneTable + 0  
0000000000200ff8  0000000000000006 R_386_GLOB_DAT 0000000000000000 __cxa_finalize + 0  
  
Relocation section '.rela.plt' at offset 4d0 contains 1 entries:  
Offset      Info          Type           Sym. Value     Sym. Name + Addend  
0000000000200fd0  0000000000000007 R_386_JMP_SLOT 0000000000000000 puts + 0  
  
-----  
Process exited after 0.3549 seconds with return value 0  
请按任意键继续. . .
```

图 2-26 C 产生重定位表

在 Elf 文件中，以“.rel”或“.rela”开头的 section 就是一个重定位段。它是一个 Elf32/64_Rela 结构数组，每个元素对应一个重定位入口。

重定位表用到的数据结构如图 2-27 所示。

```
/* Relocation table entry with addend (in section of type SHT_RELA). */
typedef struct
{
    Elf32_Addr  r_offset;           /* Address */
    Elf32_Word  r_info;            /* Relocation type and symbol index */
    Elf32_Sword r_addend;          /* Addend */
} Elf32_Rela;

typedef struct
{
    Elf64_Addr  r_offset;           /* Address */
    Elf64_Xword r_info;            /* Relocation type and symbol index */
    Elf64_Sxword r_addend;         /* Addend */
} Elf64_Rela;
```

图 2-27 数据结构

r_offset	重定位入口的偏移
r_info	表示重定位入口的类型和符号
r_addend	加上重定位尾部

表 5 具体含义

解析重定位表，与符号表相似，首先定义一个 ELF 文件头文件结构指针，得出 e_shoff，用一个节头结构指针定位到节头表，遍历节头表，对带有 .rel 字段的小节进行解析。其中的 sh_size/sh_entsize 可以得到重定位表中所包含的项的个数。sh_offset 可以得到每个节头的文件偏移。创建一个重定位表项结构指针指向节头的位置。然后根据表项个数对其进行遍历，输出每一个 r_offset 和 r_info。

其中，不同表项的 r_info 是不同的。可以用图 2-28 所示的宏进行读取。

```
#define ELF32_R_SYM(val)      ((val) >> 8)
#define ELF32_R_TYPE(val)    ((val) & 0xff)
#define ELF32_R_INFO(sym, type) (((sym) << 8) + ((type) & 0xff))

#define ELF64_R_SYM(i)       ((i) >> 32)
#define ELF64_R_TYPE(i)      ((i) & 0xffffffff)
#define ELF64_R_INFO(sym, type) (((((Elf64_Xword) (sym)) << 32) + (type))
```

图 2-28 宏

使用 ELF32/64_R_TYPE 可以得到重定位入口的类型，如图 2-29 所示。

```

#define R_386_NONE      0      /* No reloc */
#define R_386_32        1      /* Direct 32 bit */
#define R_386_PC32      2      /* PC relative 32 bit */
#define R_386_GOT32     3      /* 32 bit GOT entry */
#define R_386_PLT32     4      /* 32 bit PLT address */
#define R_386_COPY      5      /* Copy symbol at runtime */
#define R_386_GLOB_DAT  6      /* Create GOT entry */
#define R_386_JMP_SLOT  7      /* Create PLT entry */
#define R_386_RELATIVE  8      /* Adjust by program base */
#define R_386_GOTOFF    9      /* 32 bit offset to GOT */
#define R_386_GOTPC     10     /* 32 bit PC relative offset to GOT */
#define R_386_32PLT     11
#define R_386_TLS_TPOFF 14     /* Offset in static TLS block */
#define R_386_TLS_IE     15     /* Address of GOT entry for static TLS
                                block offset */
#define R_386_TLS_GOTIE  16     /* GOT entry for static TLS block
                                offset */
#define R_386_TLS_LE     17     /* Offset relative to static TLS
                                block */
#define R_386_TLS_GD     18     /* Direct 32 bit for GNU version of
                                general dynamic thread local data */
#define R_386_TLS_LDM    19     /* Direct 32 bit for GNU version of
                                local dynamic thread local data
                                in LE code */
#define R_386_16        20
#define R_386_PC16      21
#define R_386_8         22
#define R_386_PC8       23

```

图 2-29 重定位文件类型

Sym. name 与 Sym. value 可以通过头文件中的 `e_shstrndx` 得出。addend, 使用 `r_addend` 即可得出。

至此，重定位表解析完毕。

2.6 解析总结

由于时间原因，我只分析了 ELF 头文件、程序头表、节头表、符号表以及重定位表，分别对应 `readelf` 功能中的 `-h`, `-l`, `-S`, `-s` 和 `-r` 功能，我已经完成了将其存入 txt 文件中，对于其他的功能我没有进一步分析。本报告中所表明的是 64 位文件系统。32 位的和 64 位的在取字节方面有些细微的差距，就是 4 字节和 8 字节的差距，所以这个需要注意一下。但是都可以运行成功。具体代码以及运行所保存的结果（txt）可以看附带文件。

3 注入功能的实现

注入功能的实现，是指改变一个 elf 可执行文件，在其中嵌入一段代码，从而让该程序在运行之后能够先指向一个自己写的特殊的附加功能（如输出一个字符串 `helloworld`），之后再执行原来的可执行文件的功能。

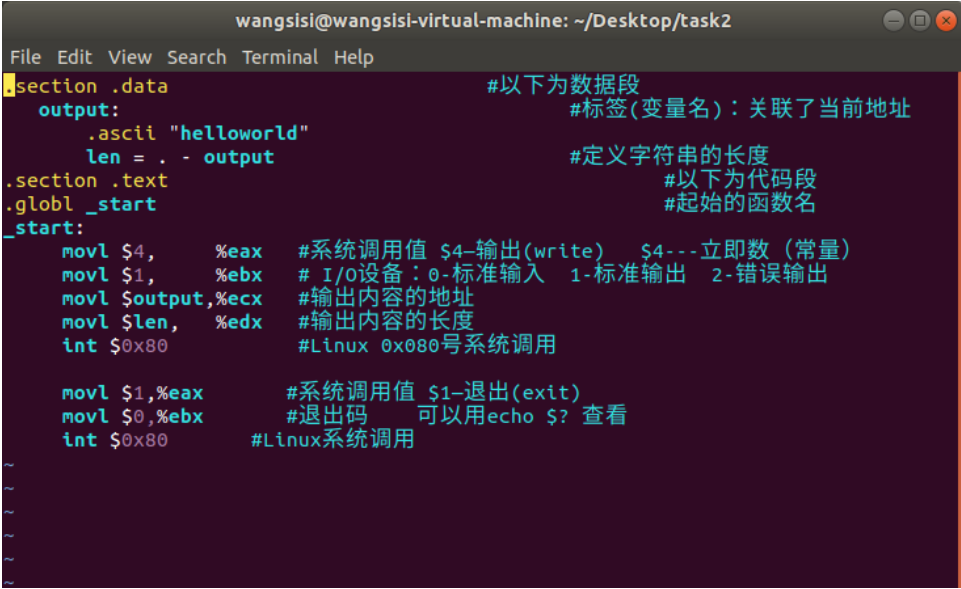
3.1 shellcode 生成

首先由于在注入前就应当了解到，因为是准备内嵌一段独立的代码，所以该段代码就不能使用其他的库，只能使用系统调用来完成它所需要的功能。在 linux 中，可以通过系统调用直接访问内核，这就决定了必须要使用汇编语言来编写附加功能。

经过查资料，我发现了这段代码叫 `shellcode`。简单的说，`Shellcode` 是一段能够完成某种特定功能的二进制代码。具体完成什么任务是由攻击者决定的，可能是开启一个新的 `shell` 或者下载某个特定的程序也或者向攻击者返回一个 `shell` 等等。也可以说，`shellcode` 就是一段不依靠 `pe` 加载器加载和处理

的(不需要重定位表、不需要导入表、不需要数据段)；在任意进程都空间都可以执行的一段二进制代码。

由于不太了解汇编，所以就去网上查阅了资料，实现了一段基本的功能。汇编代码如图 3-1 所示。

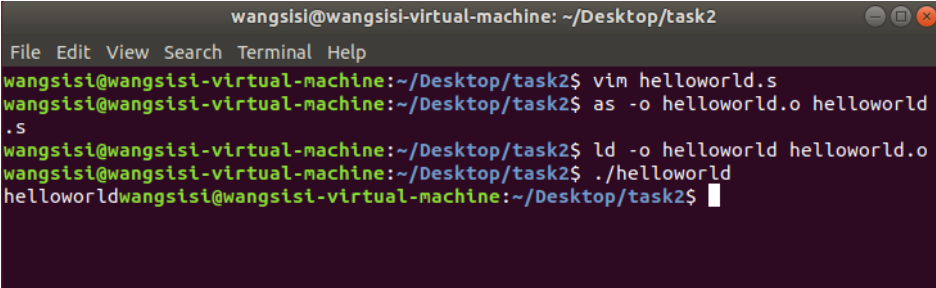


```
wangsis@wangsis-virtual-machine: ~/Desktop/task2
File Edit View Search Terminal Help
.section .data                                #以下为数据段
output:                                       #标签(变量名): 关联了当前地址
    .ascii "helloworld"
    len = . - output                         #定义字符串的长度
.section .text                                #以下为代码段
.globl _start                                #起始的函数名
_start:
    movl $4, %eax                            #系统调用值 $4-输出(write) $4---立即数(常量)
    movl $1, %ebx                            # I/O设备: 0-标准输入 1-标准输出 2-错误输出
    movl $output,%ecx                        #输出内容的地址
    movl $len, %edx                           #输出内容的长度
    int $0x80                                #Linux 0x080号系统调用

    movl $1,%eax                             #系统调用值 $1-退出(exit)
    movl $0,%ebx                             #退出码 可以用echo $? 查看
    int $0x80                                #Linux系统调用
```

图 3-1 汇编源码

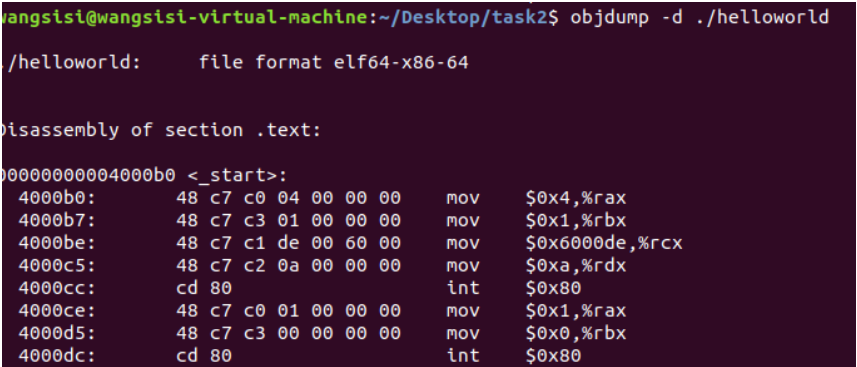
然后在本机器上运行该汇编代码，结果以及过程如图 3-2 所示。



```
wangsis@wangsis-virtual-machine: ~/Desktop/task2
File Edit View Search Terminal Help
wangsis@wangsis-virtual-machine:~/Desktop/task2$ vim helloworld.s
wangsis@wangsis-virtual-machine:~/Desktop/task2$ as -o helloworld.o helloworld.s
wangsis@wangsis-virtual-machine:~/Desktop/task2$ ld -o helloworld helloworld.o
wangsis@wangsis-virtual-machine:~/Desktop/task2$ ./helloworld
helloworldwangsis@wangsis-virtual-machine:~/Desktop/task2$
```

图 3-2 汇编运行结果

该汇编程序主要实现的一段功能就是输出一段字符串 helloworld。核心功能就是 `movl $4,%rax`，使用系统调用完成 `sys_write` 功能。如图 3-3 是机器码显示。



```
wangsis@wangsis-virtual-machine:~/Desktop/task2$ objdump -d ./helloworld
./helloworld:      file format elf64-x86-64

Disassembly of section .text:

0000000004000b0 <_start>:
4000b0: 48 c7 c0 04 00 00 00    mov     $0x4,%rax
4000b7: 48 c7 c3 01 00 00 00    mov     $0x1,%rbx
4000be: 48 c7 c1 de 00 60 00    mov     $0x6000de,%rcx
4000c5: 48 c7 c2 0a 00 00 00    mov     $0xa,%rdx
4000cc: cd 80                  int     $0x80
4000ce: 48 c7 c0 01 00 00 00    mov     $0x1,%rax
4000d5: 48 c7 c3 00 00 00 00    mov     $0x0,%rbx
4000dc: cd 80                  int     $0x80
```

图 3-3 机器码显示

至此，所需要的功能已经准备完毕，就可以得出被嵌入部分的汇编代码，其中主要的思路，首先将 `rax`, `rbx` 等寄存器 `pushq` 入栈，进行现场保存，因为注入的代码经常会变化，所以每次都需要一个栈来支持其变化。最后再 `popq` 出栈，防止对外界环境进行污染。运行完毕之后，在进行跳转指令。

因为汇编代码测试在 `ubuntu` 中总是出现段错误，无法进行测试。经过查询资料得知，所谓的段错误就是指访问的内存超过了系统所给这个程序的内存空间，通常这个值是由 `gdt` 来保存的，他是一个 48 位的寄存器，其中的 32 位是保存由它指向的 `gdt` 表，后 13 位保存相应于 `gdt` 的下标，最后 3 位包括了程序是否在内存中以及程序的在 `cpu` 中的运行级别，指向的 `gdt` 是由以 64 位为一个单位的表，在这张表中就保存着程序运行的代码段以及数据段的起始地址以及相应的断限和页面交换还有程序运行级别和内存粒度等信息，一旦一个程序发生了越界访问，`CPU` 就会产生相应的异常保护，于是 `segmentation fault` 就出现了。

即“当程序试图访问不被允许访问的内存区域（比如，尝试写一块属于操作系统的内存），或以错误的类型访问内存区域（比如，尝试写一块只读内存）。这个描述是准确的。总的来说，段错误应该就是访问了不可访问的内存，这个内存要么是不存在的，要么是受系统保护的。

由于编译器版本、函数调用约定，大小端格式、机器指令格式等各方面的因素都会对 `shellcode` 测试结果有影响。高版本的 `linux` 系统中，`shellcode` 保存在内存的 `.data` 数据段是不能被执行的（比较的低版本的 `linux` 能够执行），所以就产生了段错误。

因此，为了解决这个问题，我重新换了低版本的 `centos` 系统进行测试，最后测试成功。测试之后的汇编代码如图 3-4 所示。

```
[wangsisipromote 桌面]$ cat hello.s
.text
.global main
.type main,@function

main:
    pushq    %rax
    pushq    %rbx
    pushq    %rcx
    pushq    %rdx

    movq     $4, %rax
    movq     $1, %rbx
    movq     $output,%rcx
    movq     $10, %rdx

    popq     %rdx
    popq     %rcx
    popq     %rbx
    popq     %rax

    movq     $1,%rax
    movq     $0,%rbx
    int      $0x80
output:
    .string "helloworld"
```

图 3-4 嵌入式汇编代码

接着使用 gcc 去编译汇编文件，然后使用 objdump 进行反编译，得出了 shellcode。具体如图 3-5，图 3-6 所示。

```
wangsisipromote 桌面]$ gcc -c hello.s
wangsisipromote 桌面]$ objdump -s -d hello.o >hello.o.txt
wangsisipromote 桌面]$
```

图 3-5 反编译

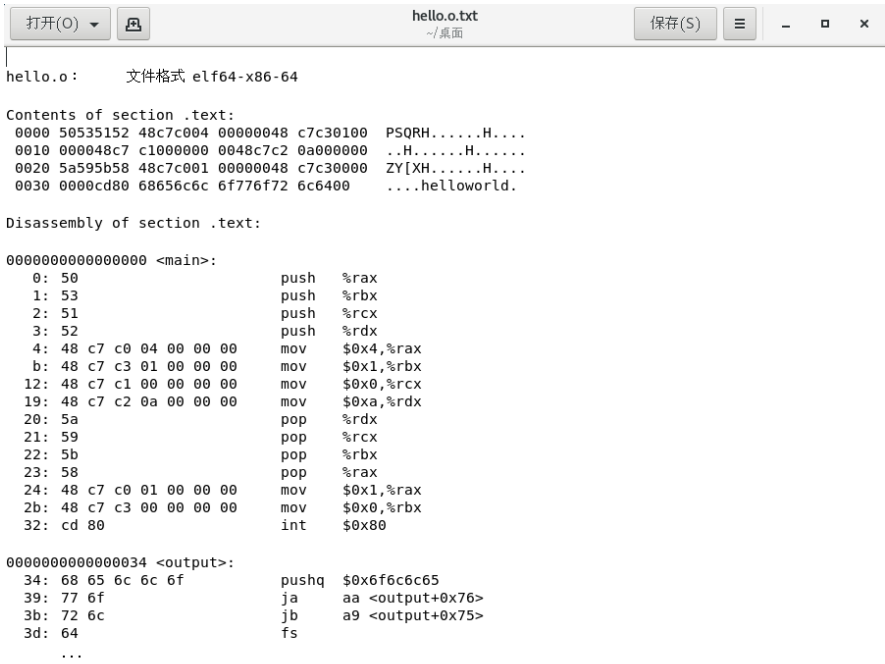


图 3-6 编译结果

这样，注入文件的 shellcode 就产生了。

3.2 注入文件

嵌入代码准备完毕，就开始注入 ELF 可执行文件了。其主要思路流程如下：1) 记录原始数据，并执行相应的提前判断

- 2) 修改程序头部表
- 3) 修改节区头部表
- 4) 计算注入机器码中相应的地址，并修改机器码
- 5) 插入机器码

其中，首先要读取 elf 头表、程序头表以及节头表，以方便保存使用。其实现的代码如表 3-1 所示：

```
// ELF Header Table 结构体
Elf64_Ehdr elf_ehdr;

// Program Header Table 结构体
Elf64_Phdr elf_phdr;

// Section Header Table 结构体
```



```
Elf64_Shdr elf_shdr;

int old_file = open(elf_file, O_RDWR);

read(old_file, &elf_ehdr, sizeof(elf_ehdr));

read(old_file, &elf_phdr, sizeof(elf_phdr));

read(old_file, &elf_shdr, sizeof(elf_shdr));
```

表 3-1 程序代码

接着便开始进行修改程序头表，因为要插入一段自己功能的机器码放在程序中，所以需要程序头表项的偏移地址增加一页(4k)，同时记录新的程序入口的虚拟地址。因为我的注入程序式插在最开始的节区，所以这里的第一个程序头表的偏移地址不需要修改，但是需要修改他的 p_filesz 和 p_memsz，由于增加了自己的注入代码，所以最后还需要 4k 对齐。其主要实现的代码如表 3-2 所示：

```
// 增加 p_offset 一页大小 4k
elf_phdr.p_offset += PAGE_SIZE;

// 寻找并更新 程序头部
lseek(old_file, elf_ehdr.e_phoff + i * elf_ehdr.e_phentsize, SEEK_SET);
write(old_file, &elf_phdr, sizeof(elf_phdr));

// 增加 p_filesz 和 p_memsz 一页大小 4k
elf_phdr.p_filesz += PAGE_SIZE;
elf_phdr.p_memsz += PAGE_SIZE;
```

表 3-2 修改程序头代码

修改完了程序头表就需要进行节区头表的修改，和修改程序头表相似，这里需要将除了第一个节区头部表外的表项中的记录的偏移地址增加一页就可以了，第一节区的表项的偏移地址不需要修改，但是需要修改他的 sh_size 节区大小，并增加 1 页 4k，因为第一个节区插入了注入代码，所以要 4k 对齐。其主要实现的代码如表 3-3 所示：

```
// 第一个节区增加一页
elf_shdr.sh_size += PAGE_SIZE;

// 节区偏移地址增加一页
elf_shdr.sh_offset += PAGE_SIZE;
```

表 3-3 修改节区头表代码

在修改完了 ELF 头表、程序头表以及节区头表之后，就开始进行插入机器码的操作了。由于要插入一个新的 ELF 可执行程序中，所以机器码中的地址需要更新成新的地址，这里采用的是数组进行保存，然后直接替换就可以了。具体的计算方法代码如表 3-4 所示。

```
// 数据段的地址, 73 为数组中程序数据段的相对位置
int data_entry = elf_ehdr.e_entry + 73;
int data_addr[4];
cal_addr(data_entry, data_addr);
```

表 3-4 机器代码地址计算

其中 cal_addr 的方法体如表 3-5 所示。

```
for (int i = 0; i < 4; i++)
{
    addr[i] = temp % 256;
    temp /= 256;
}
```

表 3-5 机器代码地址计算

这样, shellcode 执行完毕之后, 就可以执行跳转指令, 跳回 ELF 可执行文件原来的入口点。其具体的 shellcode 如图 3-7 所示。

```
char inject_code[] = {
    0x50,
    0x53,
    0x51,
    0x52,
    0x48, 0xc7, 0xc0, 0x04, 0x00, 0x00, 0x00,
    0x48, 0xc7, 0xc3, 0x01, 0x00, 0x00, 0x00,
    0x48, 0xc7, 0xc1, data_addr[0], data_addr[1], data_addr[2], data_addr[3],
    0x48, 0xc7, 0xc2, 0x0a, 0x00, 0x00, 0x00,
    0xcd, 0x80,
    0x5a,
    0x59,
    0x5b,
    0x58,
    // 此处原来的汇编程序中为程序中断指令, 修改为跳转到原入口地址elfh.e_entry
    0xbd, old_entry_addr[0], old_entry_addr[1], old_entry_addr[2], old_entry_addr[3], 0xff,
    0xe5,
    //数据区域 helloworld
    0x68, 0x65, 0x6c, 0x6c, 0x6f,
    0x77, 0x6f,
    0x72, 0x6c,
    0x64,
    0x00
};
```

图 3-7 修改后的 shellcode

机器代码整理完毕之后, 就开始插入操作。

首先保存从目标节区头到程序末尾的数据, 然后开始插入自己的机器代码, 再将机器码扩充带一页 4k, 不足的用 0 补齐。再将之前保存的数据接着注入的机器码后面插入即可。这样整个代码就注入完成了。其中主要的相关代码如表 3-6 所示。

```
// 存储原程序从节区末尾到目标节区头的数据
lseek(old_file, old_phsize, SEEK_SET);
read(old_file, data, file_stat.st_size - old_phsize);
// 插入注入代码到原 elf 文件中
```

```

lseek(old_file, old_phsize, SEEK_SET);
write(old_file, inject_code, inject_size);
// 扩充到一页
char tmp[PAGESIZE] = {0};
memset(tmp, PAGESIZE - inject_size, 0);
write(old_file, tmp, PAGESIZE - inject_size);
// 再将原始的数据接在注入代码后面插入
write(old_file, data, file_stat.st_size - old_phsize);

```

表 3-6 插入机器代码主要过程

最后进行测试，测试结果如图 3-8 所示。



```

wanglisi@localhost:~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[wanglisi@promote 桌面]$ gcc -o main main.c
[wanglisi@promote 桌面]$ gcc -o test test.c
[wanglisi@promote 桌面]$ ./test
This is the program, which will be injected.
[wanglisi@promote 桌面]$ ./main test
开始注入
开始修改程序头部表
开始修改节区头部表
开始插入注入程序
注入完成
[wanglisi@promote 桌面]$ ./test
helloworldThis is the program, which will be injected.
[wanglisi@promote 桌面]$

```

图 3-8 注入完成

3.3 注入总结

本次作业的程序注入，虽然看起来很简单，但是它涉及到了很多方面的知识，每一步出错都需要不停的翻资料，去网上查找。比如汇编语言的使用，怎么转变成机器码，机器码怎么变成 shellcode，怎么测试自己写的 shellcode 是自己想要的功能，以及 shellcode 怎么插入到注入程序当中，再插入的过程中程序运行完毕之后怎么返回到原程序。这些都需要自己思考。最让我印象深刻的就是，shellcode 测试这部分，老是出现段错误，几乎把全网翻遍了，花了好多时间，最后换了 centos 才解决。总的来说，本次大作业收获真的是很多很多了。

4 参考文献

- [1]杨新柱. 可执行文件格式分析与应用[D]. 北京邮电大学, 2009.
- [2]朱裕禄. Linux 系统下的 ELF 文件分析[J]. 电脑知识与技术, 2006 (26):111-113.
- [3]何先波, 唐宁九, 吕方, 袁敏. ELF 文件格式及应用[J]. 计算机应用研究, 2001 (11):144-145+150.
- [4]姚亚平, 蒋大明. 感染 ELF 文件病毒的技术分析[J]. 中国科技信息, 2007 (08):86-88.
- [5]傅杨, 王嘉祯, 景劼. Linux ELF 病毒感染技术研究[J]. 科协论坛(下半月), 2009 (01):61-62.
- [6]王亚刚, 陈莉君. ELF 目标文件的裁剪方法研究[J]. 电脑知识与技术, 2009, 5 (11):3018-3020.
- [7] AT&T 汇编 helloworld 实例
- [8][Linux 下 AT&T 汇编 - Hello World](#)
- [9][ELF 可执行文件的静态注入](#)
- [10][Linux 逆向---可执行文件代码静态注入小实验](#)
- [11][elf 增加一个可执行段以注入代码的一些思考](#)
- [12] [《linux 二进制分析》读书笔记总结--ELF 病毒技术-linux/unix 病毒](#)
- [13][ELF 病毒分析](#)
- [14][ELF 文件病毒的分析 and 编写](#)
- [15][汇编语言和 C 语言混合编程和内联汇编](#)
- [16][修改 ELF 可执行文件 entry 入口感染一个程序](#)
- [17][初识 shellcode](#)
- [18][黑客编程之初识 ShellCode 技术](#)
- [19][缓冲区溢出攻击实践](#)
- [20][缓冲区溢出攻击原理分析](#)
- [21][如何编写本地 shellcode](#)
- [22][hello 的 shellcode](#)
- [23][执行机器码](#)
- [24][Linux shellcode 编写入门 \(转\)](#)