使用 readelf 和 objdump 解析目标文件

一、 目标文件

目标文件的文件类型为 ELF，在 Linux 下对应文件后缀为.o 的文件，Windows 下对应文件后缀为.obj 的文件。使用 file 命令可以查看到.o 和.obj 文件均为 ELF 类型。

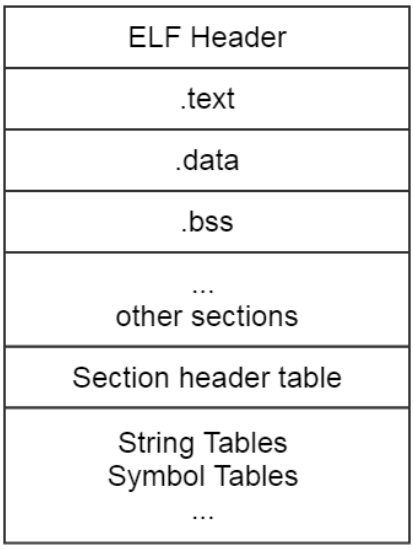
```
root@ubuntu:~/Documents/linux-c-learn/unix-learn# file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
root@ubuntu:~/Documents/linux-c-learn/unix-learn#
```

目标文件只是 ELF 文件的可重定位文件(Relocatable file)，ELF 文件一共有 4 种类型：Relocatable file、Executable file、Shared object file 和 Core Dump file{1 什么是 Core Dump：Core Dump 又叫核心转储。在程序运行过程中发生异常时，将其内存数据保存到文件中，这个过程叫做 Core Dump。2 Core Dump 的作用：在开发过程中，难免会遇到程序运行过程中异常退出的情况，这时候想要定位哪里出了问题，仅仅依靠程序自身的信息打印（日志记录）往往是不够的，这个时候就需要 Core Dump 文件来帮忙了。一个完整的 Core Dump 文件实际上相当于恢复了异常现场，利用 Core Dump 文件，可以查看到程序异常时的所有信息，变量值、栈信息、内存数据，程序异常时的运行位置（甚至记录代码行号）等等，定位所需要的一切信息都可以从 Core Dump 文件获取到，能够非常有效的提高定位效率。}

我们可以使用工具 readelf 和 objdump 对目标文件 simple.o 进行分析。为了加深对目标文件的理解，在使用 readelf & objdump 进行前，需要先要了解 ELF 文件的结构。

二、 ELF 文件结构

和 class 文件类似，ELF 文件存放数据的格式也是固定的，计算机在解析目标文件时，就是按照它每个字段的数据结构进行逐字解析的。ELF 文件结构信息定义在/usr/include/elf.h 中，整个 ELF 文件的结构如下图：



- ELF Header

ELF Header 是 ELF 文件的第一部分，64 bit 的 ELF 文件头的结构体如下：

```
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;
```

接下来我们会使用到第一个分析目标文件的工具 readelf，通过 man readelf 命令，我们可以查到 readelf 的作用就是用来显示 ELF 文件的信息。结合 Elf64_Ehdr 来看，对应解析结果如下：

```
root@ubuntu:~/Documents/linux-c-learn/unix-learn# readelf -h hello.o
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:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          320 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         12
  Section header string table index: 9
```

```
ckt@ubuntu:~/work/elf$ readelf -h simple.o
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:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          264 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```

e_ident

e_type 文件类型

e_machine

e_version

e_entry 程序的入口虚拟地址

e_phoff

e_shoff 段表在文件中的偏移

e_flags

e_ehsize ELF Header的大小

e_phentsize

e_phnum

e_shentsize 段表描述符的大小，等价于sizeof(Elf64_Ehdr)

e_shnum 拥有段的数量

e_shstrndx

- Section

完成了对 Header 的解析，再接着分析 Section 部分，Section 对应结构体如下：

```
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;
```

Section 部分主要存放的是机器指令代码和数据，执行命令 readelf -S -W hello.o 对 Section 部分的解析，解析结果和 Elf64_Shdr 也是一一对应的。

```
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00001b 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 000548 000018 18     10   1  8
  [ 3] .data             PROGBITS        0000000000000000 00005b 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 00005b 000000 00  WA  0   0  1
  [ 5] .comment          PROGBITS        0000000000000000 00005b 00002c 01  MS  0   0  1
  [ 6] .note.GNU-stack   PROGBITS        0000000000000000 000087 000000 00      0   0  1
  [ 7] .eh_frame         PROGBITS        0000000000000000 000088 000058 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000560 000030 18     10   7  8
  [ 9] .shstrtab         STRTAB          0000000000000000 0000e0 000059 00      0   0  1
  [10] .symtab           SYMTAB          0000000000000000 000440 0000f0 18     11   8  8
  [11] .strtab           STRTAB          0000000000000000 000530 000012 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (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)
```

对于这部分内容，通常我们比较的 Section 是**.text（存放代码）**、**.data（存放全局静态变量和局部静态变量）**和**.bss（存未初始化的全局变量和局部静态变量）**，在后面会对这几个段分别分进行解析。

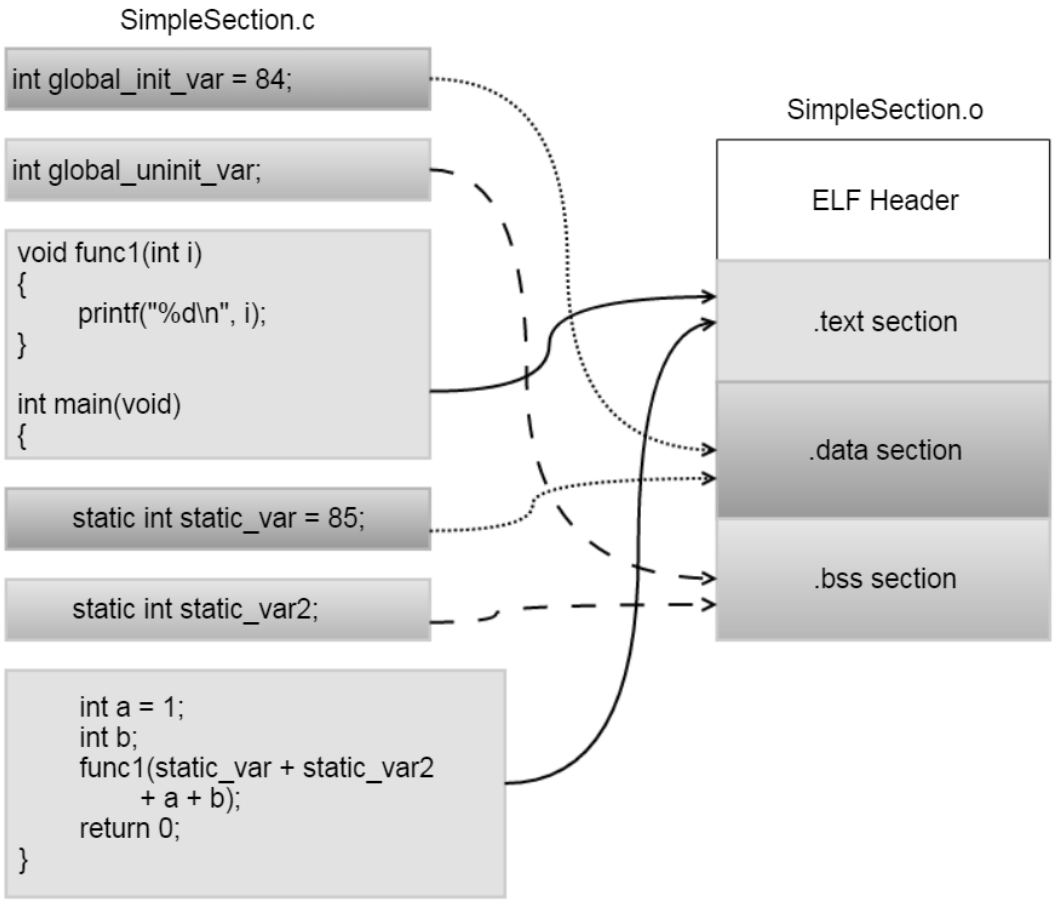
解析目标文件

分析完 ELF 文件结构，接着来解析一个目标文件。首先，准备好源码 SimpleSection.c，执行命令 gcc -c SimpleSection.c 生成目标文件 SimpleSection.o。

```
int printf(const char* format, ...);

int global_init_var = 84;
int global_uninit_var;

void func1(int i)
{
    printf("%d\n", i);
}

int main(void)
{
    static int static_var = 85;
    static int static_var2;
    int a = 1;
    int b;
    func1(static_var + static_var2 + a + b);
    return 0;
}
```

执行命令 objdump -h SimpleSection.o 对 Section 部分进行解析，我们可以得到每个段的大小。

```
root@ubuntu:~/Documents/linux-c-learn/unix-learn# objdump -h SimpleSection.o

SimpleSection.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000056  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  0000000000000000  0000000000000000  00000098  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000a0  2**2
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000002c  0000000000000000  0000000000000000  000000a4  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000d0  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000d0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```



我们的代码是存放到.text 中，已初始化全局变量和局部静态变量存放在.data 中，未初始化全局变量和局部静态变量存放在.bss 中。

执行命令 **objdump -s -d SimpleSection.o** 对代码段(.text)的解析结果如下：

```
Contents of section .text:
 0000 554889e5 4883ec10 897dfc8b 45fc89c6  UH..H....}..E...
 0010 bf000000 00b80000 0000e800 000000c9  ................
 0020 c3554889 e54883ec 10c745f8 01000000  .UH..H....E.....
 0030 8b150000 00008b05 00000000 01c28b45  ...............E
 0040 f801c28b 45fc01d0 89c7e800 000000b8  ....E...........
 0050 00000000 c9c3                        ......
Contents of section .data:
 0000 54000000 55000000                    T...U...
Contents of section .rodata:
 0000 25640a00                             %d..
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520342e  .GCC: (Ubuntu 4.
 0010 382e342d 32756275 6e747531 7e31342e  8.4-2ubuntu1~14.
 0020 30342e34 2920342e 382e3400           04.4) 4.8.4.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 21000000 00410e10 8602430d  ....!....A....C.
 0030 065c0c07 08000000 1c000000 3c000000  .\..........<...
 0040 00000000 35000000 00410e10 8602430d  ....5....A....C.
 0050 06700c07 08000000                    .p......

Disassembly of section .text:

0000000000000000 <func1>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   8b 45 fc                mov    -0x4(%rbp),%eax
   e:   89 c6                   mov    %eax,%esi
  10:   bf 00 00 00 00          mov    $0x0,%edi
  15:   b8 00 00 00 00          mov    $0x0,%eax
  1a:   e8 00 00 00 00          callq  1f <func1+0x1f>
  1f:   c9                      leaveq
  20:   c3                      retq

0000000000000021 <main>:
  21:   55                      push   %rbp
  22:   48 89 e5                mov    %rsp,%rbp
  25:   48 83 ec 10             sub    $0x10,%rsp
  29:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
  30:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 36 <main+0x15>
  36:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 3c <main+0x1b>
  3c:   01 c2                   add    %eax,%edx
  3e:   8b 45 f8                mov    -0x8(%rbp),%eax
  41:   01 c2                   add    %eax,%edx
  43:   8b 45 fc                mov    -0x4(%rbp),%eax
  46:   01 d0                   add    %edx,%eax
  48:   89 c7                   mov    %eax,%edi
  4a:   e8 00 00 00 00          callq  4f <main+0x2e>
  4f:   b8 00 00 00 00          mov    $0x0,%eax
  54:   c9                      leaveq
  55:   c3                      retq
```

执行命令 **objdump -s -d SimpleSection.o** 对数据段和只读数据段解析结果如下：

执行命令 objdump -x -s -d SimpleSection.o 打印出目标文件的符号表：