测试ELF测试用代码和so.note

简介

首先，你需要知道的是所谓对象文件(Object files)有三个种类：

1）可重定位的对象文件(Relocatable file)

这是由汇编器汇编生成的 .o 文件。后面的链接器(link editor)拿一个或一些 Relocatable object files 作为输入，经链接处理后，生成一个可执行的对象文件(Executable file) 或者一个可被共享的对象文件(Shared object file)。我们可以使用 ar 工具将众多的 .o Relocatable object files 归档(archive)成 .a 静态库文件。如何产生 Relocatable file，你应该很熟悉了，请参见我们相关的基本概念文章和JulWiki。另外，可以预先告诉大家的是我们的内核可加载模块 .ko 文件也是 Relocatable object file。

2) 可执行的对象文件(Executable file)

这我们见的多了。文本编辑器vi、调式用的工具gdb、播放mp3歌曲的软件mplayer等等都是Executable object file。你应该已经知道，在我们的 Linux 系统里面，存在两种可执行的东西。除了这里说的 Executable object file，另外一种就是可执行的脚本(如shell脚本)。注意这些脚本不是 Executable object file，它们只是文本文件，但是执行这些脚本所用的解释器就是 Executable object file，比如 bash shell 程序。

3) 可被共享的对象文件(Shared object file)

这些就是所谓的动态库文件，也即 **.so** 文件。如果拿前面的静态库来生成可执行程序，那每个生成的可执行程序中都会有一份库代码的拷贝。如果在磁盘中存储这些可执行程序，那就会占用额外的磁盘空间；另外如果拿它们放到Linux系统上一起运行，也会浪费掉宝贵的物理内存。

如果将静态库换成动态库，那么这些问题都不会出现。动态库在发挥作用的过程中，必须经过两个步骤：

a）链接编辑器(link editor)拿它和其他Relocatable object file以及其他shared object file作为输入，经链接处理后，生成另外的 shared object file 或者 executable file。

b）在运行时，动态链接器(dynamic linker)拿它和一个Executable file以及另外一些 Shared object file 来一起处理，在Linux系统里面创建一个进程映像。

```
yananhdeMacBook-Pro:armeabi-v7a yananh$ file libtest.so
libtest.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, BuildID[sha1]=9363ce0db74cb9979d09fe651a98f78581c8b966, stripped
yananhdeMacBook-Pro:armeabi-v7a yananh$
```

ELF文件格式

对静态链接与动态链接有基本的了解之后，我们需要明白，目标文件得按照一定的格式来组织，以便链接器获取足够的信息完成这两种链接过程。当然，目标文件的格式不止限于链接这种作用。可执行文件在被执行之前需要进行装载，操作系统要友好的将其载入到进程的地址空间中，同样，可执行文件也得按照一定格式组织。

前面我们了解了所谓的链接，即读取目标文件中的相关信息解决符号跨模块之间的引用问题。那么执行是个什么基本过程了？《程序员的自我修养》中讲到，很多时候一个程序被执行同时都伴随着一个新的进程的创建，那么最通常的情形便是：创建一个进程，然后装载相应的可执行文件并且执行。在有虚拟存储的情况下，上述过程的最开始只需要做三件事情：

- 创建一个独立的虚拟地址空间；
- 读取可执行文件头，并且建立虚拟空间与可执行文件的映射关系；
- 将CPU的指令寄存器设置成可执行文件的入口地址，启动运行；

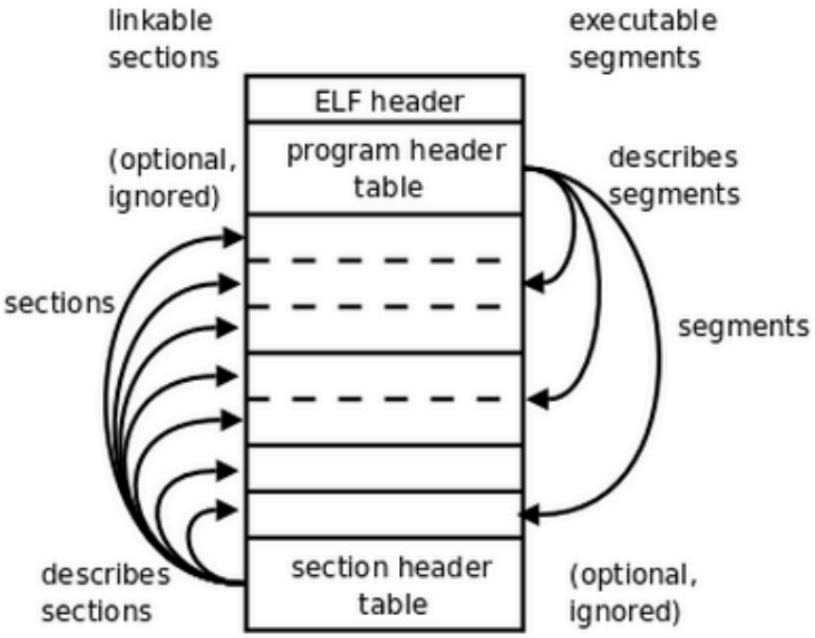
第一步完成了虚拟空间到物理内存的映射关系，而第二步专注于虚拟空间与可执行文件的映射关系。这样，当程序执行发生页错误时，操作系统知道程序当前所需要的页在可执行文件中的哪一个

位置。从某种角度来说，第二步应该是整个装载过程中最重要的一步。那么可执行文件到底是怎样与虚拟空间映射起来的了？

ELF链接视图和执行视图

我们先了解下可执行文件结构，然后尝试回答上面的问题！

可执行文件的设计者们则将目标文件的格式与可执行文件的格式进行了统一，均为ELF格式文件，该文件格式提供了两种视图，如下，是ELF文件的基本格式，左边是链接视图，右边是执行视图。



链接视图是以节（section）为单位，执行视图是以段（segment）为单位。

链接视图就是在链接时用到的视图，而执行视图则是在执行时用到的视图。上图左侧的视角是从链接来看的，右侧的视角是执行来看的。

总个文件可以分为四个部分：

- ELF header： 描述整个文件的组织。
- Program Header Table: 描述文件中的各种segments，用来告诉系统如何创建进程映像的。
- Section Header Table: 包含了文件各个segction的属性信息，我们都将结

合例子来解释。

- sections 或者 segments：segments是从运行的角度来描述elf文件，sections是从链接的角度来描述elf文件，也就是说，在链接阶段，我们可以忽略program header table来处理此文件，在运行阶段可以忽略section header table来处理此程序（所以很多加固手段删除了section header table）。从图中我们也可以看出，segments与sections是包含的关系，一个segment包含若干个section。

- sections 或者 segments：segments是从运行的角度来描述elf文件，sections是从链接的角度来描述elf文件，也就是说，在链接阶段，我们可以忽略program header table来处理此文件，在运行阶段可以忽略section header table来处理此程序（所以很多加固手段删除了section header table）。从图中我们也可以看出，segments与sections是包含的关系，一个segment包含若干个section。
- Section Header Table: 包含了文件各个segction的属性信息，我们都将结合例子来解释。

程序头部表（Program Header Table），如果存在的话，告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区头部表（Section Header Table），包含了描述文件节区的信息，比如大小、偏移等。每个节区在表中 都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包 含节区头部表，其他目标文件可以有，也可以没有这个表。

需要注意地是：尽管图中显示的各个组成部分是有顺序的，实际上除了 ELF 头部表以外，其他节区和段都没有规定的顺序。

右半图是以程序执行视图来看待的，与左边对应，多了一个段（segment）的概念，编译器在生成目标文件时，通常使用从零开始的相对地址，而在链接过程中，链接器从一个指定的地址开始，根据输入目标文件的顺序，以段（segment）为单位将它们拼装起来。其中每个段可以包括很多个节（section）。

使用010Editor通过ELF模板打开libtest.so格式如下：

Name	Value	Start	Size	Color	Comment
struct file		0h	2F0h	Fg: Bg:	
struct elf_header		0h	34h	Fg: Bg:	The main elf header basically tells us where everything is loc...
struct program_header_table		34h	100h	Fg: Bg:	Program headers - describes the sections of the program th...
struct section_header_table		2284h	3E8h	Fg: Bg:	
struct dynamic_symbol_table		1F0h	100h	Fg: Bg:	

ELF的数据类型定义

在具体介绍ELF的格式之前，我们先来了解在ELF文件中都有哪些数据类型的定

义：

名称	大小	对齐	目的
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等整数
Elf32_Off	4	4	无符号文件偏移
Elf32_SWord	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

文件头格式

1.2 Sections

在101Editor中能够获取所有的section。

Name	Value	Start	Size	Color	Comment
▼ struct file		0h	2F0h	Fg: Bg:	
▶ struct elf_header		0h	34h	Fg: Bg:	The main elf header basically tells us where everything is loc...
▶ struct program_header_table		34h	100h	Fg: Bg:	Program headers - describes the sections of the program th...
▼ struct section_header_table		2284h	3E8h	Fg: Bg:	
▶ struct section_table_entry32...	SHN_UNDEF	2284h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.note.android.ident	22ACh	28h	Fg: Bg:	
▶ struct section_table_entry32...	.note.gnu.build-id	22D4h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.dynsym	22FCh	28h	Fg: Bg:	
▶ struct section_table_entry32...	.dynstr	2324h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.hash	234Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.gnu.version	2374h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.gnu.version_d	239Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.gnu.version_r	23C4h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.rel.dyn	23ECh	28h	Fg: Bg:	
▶ struct section_table_entry32...	.rel.plt	2414h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.plt	243Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.text	2464h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.ARM.extab	248Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.ARM.exidx	24B4h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.rodata	24DCh	28h	Fg: Bg:	
▶ struct section_table_entry32...	.fini_array	2504h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.dynamic	252Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.got	2554h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.data	257Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.bss	25A4h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.comment	25CCh	28h	Fg: Bg:	
▶ struct section_table_entry32...	.note.gnu.gold-version	25F4h	28h	Fg: Bg:	
▶ struct section_table_entry32...	.ARM.attributes	261Ch	28h	Fg: Bg:	
▶ struct section_table_entry32...	.shstrtab	2644h	28h	Fg: Bg:	
▶ struct dynamic_symbol_table		1F0h	100h	Fg: Bg:	

通过readelf -S libtest.so查看文件中所有的section

```
yananhdeMacBook-Pro:armeabi-v7a yananh$ readelf -S libtest.so
There are 25 section headers, starting at offset 0x2284:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.android.ide NOTE            00000134 000134 000098 00   A  0   0  4
  [ 2] .note.gnu.build-i NOTE            000001cc 0001cc 000024 00   A  0   0  4
  [ 3] .dynsym           DYNSYM          000001f0 0001f0 000100 10   A  4   1  4
  [ 4] .dynstr           STRTAB          000002f0 0002f0 0000ec 00   A  0   0  1
  [ 5] .hash             HASH            000003dc 0003dc 000054 04   A  3   0  4
  [ 6] .gnu.version      VERSYM          00000430 000430 000020 02   A  3   0  2
  [ 7] .gnu.version_d    VERDEF          00000450 000450 00001c 00   A  4   1  4
  [ 8] .gnu.version_r    VERNEED         0000046c 00046c 000020 00   A  4   1  4
  [ 9] .rel.dyn          REL             0000048c 00048c 000048 08   A  3   0  4
  [10] .rel.plt          REL             000004d4 0004d4 000050 08  AI  3  18  4
  [11] .plt              PROGBITS        00000524 000524 00008c 00  AX  0   0  4
  [12] .text             PROGBITS        000005b0 0005b0 0015a4 00  AX  0   0  4
  [13] .ARM.extab        PROGBITS        00001b54 001b54 00003c 00   A  0   0  4
  [14] .ARM.exidx        ARM_EXIDX       00001b90 001b90 000100 08  AL 12   0  4
  [15] .rodata           PROGBITS        00001c90 001c90 00000a 01 AMS  0   0  1
  [16] .fini_array       FINI_ARRAY      00002ea4 001ea4 000004 04  WA  0   0  4
  [17] .dynamic          DYNAMIC         00002ea8 001ea8 000108 08  WA  4   0  4
  [18] .got              PROGBITS        00002fb0 001fb0 000050 00  WA  0   0  4
  [19] .data             PROGBITS        00003000 002000 000004 00  WA  0   0  4
  [20] .bss              NOBITS          00003004 002004 000000 00  WA  0   0  1
  [21] .comment          PROGBITS        00000000 002004 00012f 01  MS  0   0  1
  [22] .note.gnu.gold-ve NOTE            00000000 002134 00001c 00      0   0  4
  [23] .ARM.attributes   ARM_ATTRIBUTES  00000000 002150 000036 00      0   0  1
  [24] .shstrtab         STRTAB          00000000 002186 0000fe 00      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),
  y (noread), p (processor specific)
```