ELF section 修复的一些思考

一、概述

相信各位读者对 so 分析都采用静态和动态相结合的方式，静态分析常用 readelf、objdump、ida 等工具，这些工具对 so 文件的分析都会使用到 Section 信息。从这篇帖子中 http://bbs.pediy.com/showthread.php?t=191649 知道，程序并不需要 section 信息。现很多 so 文件对 section 信息都进行了处理，导致常用分析工具无法使用。以下讨论前段时间对 section 修复的一些思考，若有不足或错误之处，请各位大大指正，小弟感激不尽！

二、仅处理 so 文件头

在上文提到的帖子中，给出了一种 section 处理的一种简单方式。这里在罗嗦下，即将 Elf32_Ehdr 中的 e_shoff, e_shnum, e_shstrndx, e_shentsize 字段处理。修复公式：e_shstrndx = e_shnum -1; e_shnum = (file_size – e_shoff) / sizeof(Elf32_Shdr)。在那篇帖子中作为修复的数字 so 文件，并未处理 e_shoff 字段，故用上式修复可行。那如果都处理掉，则上式中存在两个未知数，无法利用。

一种简单的思路是，手动查找 so 文件中一些稳定且标志性的数据作为参考来修复。这里，我选择 shstrtab 表，这样比较简单。因为 shshtrtab 后面就是 section 头信息，这样就间接找到 e_shoff 位置，即可利用上式修复。

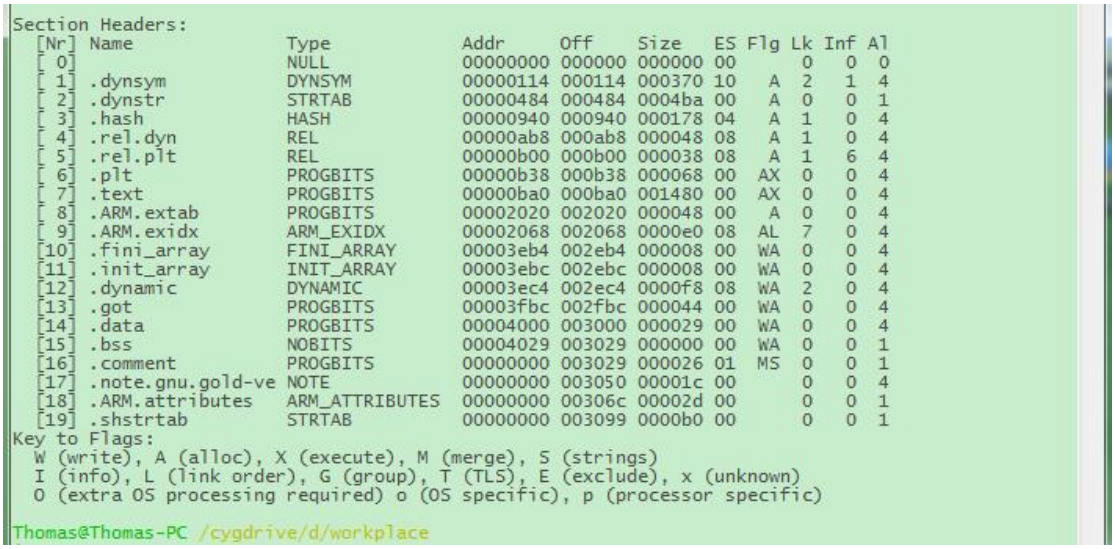
手动查找当然可行，毕竟麻烦。作为程序猿，应该通过程序来解决问题。借鉴手动修复的思路，只要程序能找到 shstrtab 即可实现修复。从观察或 e_shstrndx 知道，shstrtab section 为最后一个 section，即处于文件末尾。那直接移动到末尾读取到 shstrtab section，则 e_shoff = sh_offset + sh_zize(这里还需对 e_shoff 4 字节对齐处理)。

三、无 section 信息

现阶段，我遇到的很多 so 文件的 section 修复都采用上述方法，还未遇到无 section 信息的 so，即直接将 so 文件中的 section 直接删除，或者替换 section 内容(比如填充隐藏代码或者垃圾数据之类的)。直接删除 section 信息，即可节省空间，又可让静态工具“蛋疼”(有点好奇为什么不对 so 作如此处理)。另外，直接从内存中 dump 出来的 so 文件，也是没有 section 信息的(因为 section 没有被加载到内存中)。当然，从内存中 dump 的 so 文件，由于内存对齐的原因，需要作下简单处理，这里就不赘述。

从内存 dump 出的 so 文件已经经过解密，直接拿来分析，能获得事半功倍的效果。但没有 section 信息，静态分析很是不爽。如果原 so 文件有 section 信息，则只需要把原 so 文件中的 section 信息复制过来并修复 Elf32_Ehdr 即可。那如果原 so 无 section 信息，我的理解是需要对 section 信息进行重建(虽然现阶段还没用到，处于问题思考的完整性，讨论这种情况)。

使用 readelf –S 查看一个完整的 so 文件 section 如下图所示：

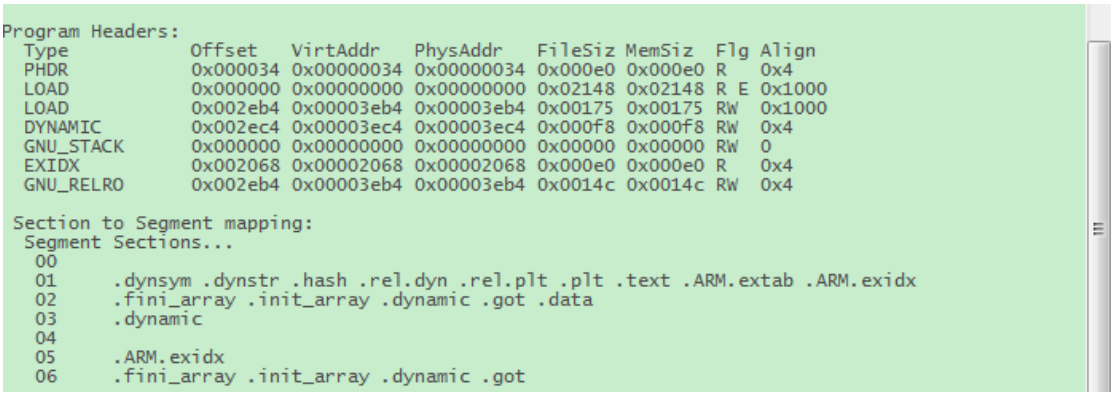


```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .dynsym           DYNSYM          00000114 000114 000370 10   A  2   1  4
  [ 2] .dynstr           STRTAB          00000484 000484 0004ba 00   A  0   0  1
  [ 3] .hash             HASH            00000940 000940 000178 04   A  1   0  4
  [ 4] .rel.dyn          REL             00000ab8 000ab8 000048 08   A  1   0  4
  [ 5] .rel.plt          REL             00000b00 000b00 000038 08   A  1   6  4
  [ 6] .plt              PROGBITS        00000b38 000b38 000068 00  AX  0   0  4
  [ 7] .text             PROGBITS        00000ba0 000ba0 001480 00  AX  0   0  4
  [ 8] .ARM.extab        PROGBITS        00002020 002020 000048 00   A  0   0  4
  [ 9] .ARM.exidx        ARM_EXIDX       00002068 002068 0000e0 08  AL  7   0  4
  [10] .fini_array       FINI_ARRAY      00003eb4 002eb4 000008 00  WA  0   0  4
  [11] .init_array       INIT_ARRAY      00003ebc 002ebc 000008 00  WA  0   0  4
  [12] .dynamic          DYNAMIC         00003ec4 002ec4 0000f8 08  WA  2   0  4
  [13] .got              PROGBITS        00003fbc 002fbc 000044 00  WA  0   0  4
  [14] .data             PROGBITS        00004000 003000 000029 00  WA  0   0  4
  [15] .bss              NOBITS          00004029 003029 000000 00  WA  0   0  1
  [16] .comment          PROGBITS        00000000 003029 000026 01  MS  0   0  1
  [17] .note.gnu.gold-ve NOTE            00000000 003050 00001c 00      0   0  4
  [18] .ARM.attributes   ARM_ATTRIBUTES  00000000 00306c 00002d 00      0   0  1
  [19] .shstrtab         STRTAB          00000000 003099 0000b0 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Thomas@Thomas-PC /cygdrive/d/workplace
```

图 1

使用 readelf –l 如图所示：



```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x000e0 0x000e0 R   0x4
  LOAD           0x000000 0x00000000 0x00000000 0x02148 0x02148 R E 0x1000
  LOAD           0x002eb4 0x00003eb4 0x00003eb4 0x00175 0x00175 RW  0x1000
  DYNAMIC        0x002ec4 0x00003ec4 0x00003ec4 0x000f8 0x000f8 RW  0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0
  EXIDX          0x002068 0x00002068 0x00002068 0x000e0 0x000e0 R   0x4
  GNU_RELRO      0x002eb4 0x00003eb4 0x00003eb4 0x0014c 0x0014c RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .dynsym .dynstr .hash .rel.dyn .rel.plt .plt .text .ARM.extab .ARM.exidx
   02     .fini_array .init_array .dynamic .got .data
   03     .dynamic
   04
   05     .ARM.exidx
   06     .fini_array .init_array .dynamic .got
```

图 2

从 segment 信息可以看出, 对.dynamic 和.arm_exidx 的 section 重建很简单，即读取即可。通过.dynamic，可以对大部分 section 进行重建，具体如下：

1. 通过 DT_SYMTAB，DT_STRTAB，DT_STRSZ，DT_REL，DT_RELSZ，DT_JMPREL，DT_PLTRELSZ，DT_INIT_ARRAY，DT_INIT_ARRAYSZ，DT_FINI_ARRAY，DT_FINI_ARRAYSZ 得到.dynsym，.dynstr, rel.dyn, rel.plt, init_array, fini_array 相应的 section vaddr 和 size 信息，完成对上述 section 的重建。这里需要注意，处于 load2 中的 section，offset = vaddr – 0x1000
2. 通过 DT_HASH 得到 hash section 的 vaddr，然后读入前两项得到 nbucket 和 nchain 的值，得到 hashsz = (nbucket + nchain + 2) * sizeof(int), 完成对 hash 表重建
3. Plt 的起始位置即为 rel.plt 的末尾，通过 1 中的对 rel.plt 的处理，即可得到 plt 的 offset 和 vaddr 信息。通过 plt 的结构知道，plt 由固定 16 字节 + 4 字节的__global__offset_table 变量和 n 个需要重定位的函数地址构成，函数地址又与 rel.plt 中的结构一一对应。故 size = (20 + 12 * (rel.plt.size) / sizeof(Elf32_Rel)。
4. 从 DT_PLTGOT 可以得到__global_offset_table 的偏移位置。由 got 表的结构知道，__global_offset_table 前是 rel.dyn 重定位结构，之后为 rel.plt 重定位结构，都与 rel 一一对应。则 got 表的重建具体为：通过已重建的.dynamic 得到 got 起始位置，通过__global_offset_table 偏移 + 4 * (rel.plt.size) / sizeof(Elf32_Rel)(这里还需要添加 2 个 int 的填充位置)得到 got 的末尾，通过首尾位置得到 got 的 size，完成重建

5. 通过 got 的末尾，得到 data 的起始位置，再通过 load2_vaddr + load2_filesz 得到 load2 的末尾(load2 即第二个 LOAD)，即 data 的末尾位置，计算长度，完成修正。可能读者会问，bss 才是 load2 的最后一个 section。的确，但 bss 为 NOBITS，即可把 data 看作 load2 最后一个 section。
6. 对 bss 的修正就很简单，offset 和 vaddr 即为 load2 末尾。由于未 NOBITS 类型，长度信息无关紧要。
7. 到这里，读者可能已经发现，还没对 text 和 ARM.extab 修正。限于本人水平，还没能找到方法区分开这两个 section。现处理是将之合并，作为 text & ARM.extab 节。具体修正：offset 和 vaddr 通过 plt 末尾得到，长度通过 ARM.exidx 的起始位置和 plt 末尾位置计算得到。

至此，绝大部分 section 信息已经重建完成。最后，在将 shstrtab 添加，并修正 Elf32_Ehdr，完成 section 重建。虽然未 100%重建，但已经能够帮助分析了。重建后的如图所示，图中红色部分即是未分离的 test & ARM.extab section。

```
There are 16 section headers, starting at offset 0x30b0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .dynsym           DYNSYM          00000114 000114 000370 10   A  2   1  4
  [ 2] .dynstr           STRTAB          00000484 000484 0004ba 00   A  0   0  1
  [ 3] .hash             HASH            00000940 000940 000178 04   A  4   1  4
  [ 4] .rel.dyn          REL             00000ab8 000ab8 000048 08   A  1   0  4
  [ 5] .rel.plt          REL             00000b00 000b00 000038 08   A  1   6  4
  [ 6] .plt              PROGBITS        00000b38 000b38 000068 00  AX  0   0  4
  [ 7] .text&.ARM.extab  PROGBITS        00000ba0 000ba0 0014c8 00  AX  0   0  4
  [ 8] .ARM.exidx        ARM_EXIDX       00002068 002068 0000e0 08  AL  7   0  4
  [ 9] .fini_array       FINI_ARRAY      00003eb4 002eb4 000008 00  WA  0   0  4
  [10] .init_array       INIT_ARRAY      00003ebc 002ebc 000008 00  WA  0   0  4
  [11] .dynamic          DYNAMIC         00003ec4 002ec4 0000f8 08  WA  2   0  4
  [12] .got              PROGBITS        00003fbc 002fbc 000044 00  WA  0   0  4
  [13] .data             PROGBITS        00004000 003000 000029 00  WA  0   0  4
  [14] .bss              NOBITS          00004029 003029 000000 00  WA  0   0  1
  [15] .shstrtab         STRTAB          00000000 003029 000085 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Thomas@Thomas-PC /cygdrive/d/workplace
```

图 3

使用 ida 也能正常打开，只是会将 ARM.extab 的数据转换成错误代码，其他均正常。

```
.text&.ARM.extab:0000201C                   BL              abort
.text&.ARM.extab:00002020 ; ---------------------------------------------------------------------------
.text&.ARM.extab:00002020                   TSTHI           R1, R8,LSL#2
.text&.ARM.extab:00002024                   STRHI           R11, [R0],#-0xB0
.text&.ARM.extab:00002028                   ANDEQ           R0, R0, R0
.text&.ARM.extab:0000202C                   SMLATBHI        R1, R1, R2, R11
.text&.ARM.extab:00002030                   STREQH          R11, [LR,R0]!
.text&.ARM.extab:00002034                   ANDEQ           R0, R0, R0
.text&.ARM.extab:00002038                   TSTHI           R1, PC,LSR R6
.text&.ARM.extab:0000203C                   LDRHIB          R11, [PC],#-0xB0
.text&.ARM.extab:00002040                   ANDEQ           R0, R0, R0
.text&.ARM.extab:00002044                   TSTHI           R1, R8,LSL#2
.text&.ARM.extab:00002048                   STRHI           R11, [R0],#-0xB0
.text&.ARM.extab:0000204C                   ANDEQ           R0, R0, R0
.text&.ARM.extab:00002050                   TSTHI           R1, R8,LSL#2
.text&.ARM.extab:00002054                   STRHI           R11, [R0],#-0xB0
.text&.ARM.extab:00002058                   ANDEQ           R0, R0, R0
.text&.ARM.extab:0000205C                   TSTHI           R1, R8,LSL#2
.text&.ARM.extab:00002060                   STRHI           R11, [R0],#-0xB0
.text&.ARM.extab:00002064                   ANDEQ           R0, R0, R0
.text&.ARM.extab:00002064 ; End of function _Unwind_GetTextRelBase
.text&.ARM.extab:00002064
.text&.ARM.extab:00002064 ; .text_.ARM.extab ends
.text&.ARM.extab:00002064
```