|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1. UNIX/LINUX 平台可执行文件格式分析   http://www.ibm.com/i/c.gif | developerWorks | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| http://www.ibm.com/i/c.gif | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | http://www.ibm.com/i/c.gif | |  | | --- | | **文档选项** |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | http://www.ibm.com/i/c.gif | 将打印机的版面设置成横向打印模式 | [**打印本页**](javascript:print()) | | http://www.ibm.com/i/c.gif | 将此页作为电子邮件发送 | [**将此页作为电子邮件发送**](javascript:document.email.submit();) | | |  |  |  |  | | --- | --- | --- | | http://www.ibm.com/i/c.gif | | | |  | http://www.ibm.com/i/c.gif | 施聪，成都人，高级程序员、网络设计师。长期从事基于UNIX/LINUX下的c/c++程序设计和数据库建模工作。可通过 [javer@163.com](mailto:javer@163.com?cc=javer@163.com)或 [memncmp@yahoo.com.cn](mailto:memncmp@yahoo.com.cn?cc=javer@163.com)和他联系。 | |

级别： 初级

[施聪](http://www.ibm.com/developerworks/cn/linux/l-excutff/#author) ([memncmp@yahoo.com.cn](mailto:memncmp@yahoo.com.cn?subject=UNIX/LINUX%20平台可执行文件格式分析&cc=javer@163.com)), 高级程序员、网络设计师

2004 年 12 月 01 日

本文讨论了 UNIX/LINUX 平台下三种主要的可执行文件格式：**a.out**（assembler and link editor output 汇编器和链接编辑器的输出）、**COFF**（Common Object File Format 通用对象文件格式）、**ELF**（Executable and Linking Format 可执行和链接格式）。首先是对可执行文件格式的一个综述，并通过描述 ELF 文件加载过程以揭示可执行文件内容与加载运行操作之间的关系。随后依此讨论了此三种文件格式，并着重讨论 ELF 文件的动态连接机制，其间也穿插了对各种文件格式优缺点的评价。最后对三种可执行文件格式有一个简单总结，并提出作者对可文件格式评价的一些感想。

**可执行文件格式综述**

相对于其它文件类型，可执行文件可能是一个操作系统中最重要的文件类型，因为它们是完成操作的真正执行者。可执行文件的大小、运行速度、资源占用情况以及可扩展性、可移植性等与文件格式的定义和文件加载过程紧密相关。**研究可执行文件的格式对编写高性能程序和一些黑客技术的运用都是非常有意义的。**

不管何种可执行文件格式，一些基本的要素是必须的，显而易见的，文件中应包含代码和数据。因为文件可能引用外部文件定义的符号（变量和函数），因此重定位信息和符号信息也是需要的。一些辅助信息是可选的，如调试信息、硬件信息等。基本上任意一种可执行文件格式都是按区间保存上述信息，称为段（Segment）或节（Section）。不同的文件格式中段和节的含义可能有细微区别，但根据上下文关系可以很清楚的理解，这不是关键问题。最后，可执行文件通常都有一个文件头部以描述本文件的总体结构。

相对可执行文件有三个重要的概念：编译（compile）、连接（link，也可称为链接、联接）、加载（load）。源程序文件被编译成目标文件，多个目标文件被连接成一个最终的可执行文件，可执行文件被加载到内存中运行。因为本文重点是讨论可执行文件格式，因此加载过程也相对重点讨论。下面是LINUX平台下ELF文件加载过程的一个简单描述。

1：**内核首先读ELF文件的头部，**然后根据头部的数据指示分别读入各种数据结构，**找到标记为可加载（loadable）的段**，并调用函数 mmap()把段内容加载到内存中。**在加载之前，内核把段的标记直接传递给 mmap()**，**段的标记指示该段在内存中是否可读、可写，可执行。**显然，文本段是只读可执行，而数据段是可读可写。***这种方式是利用了现代操作系统和处理器对内存的保护功能。***著名的Shellcode（ [参考资料 17](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)）的编写技巧则是突破此保护功能的一个实际例子。

2：内核分析出**ELF文件标记为 PT\_INTERP 的段**中所对应的动态连接器名称，并加载动态连接器。现代 LINUX 系统的动态连接器通常是 /lib/ld-linux.so.2，相关细节在后面有详细描述。

3：**内核在新进程的堆栈中设置一些标记-值对，以指示动态连接器的相关操作。**

4：内核把控制传递给动态连接器。

5：动态连接器检查程序对外部文件（共享库）的依赖性，**并在需要时对其进行加载。**

6：动态连接器对程序的外部引用进行重定位，通俗的讲，就是告诉程序其引用的外部变量/函数的地址，**此地址位于共享库被加载在内存的区间内**。动态连接还有一个延迟（Lazy）定位的特性，即只在"真正"需要引用符号时才重定位，这对提高程序运行效率有极大帮助。

7：动态连接器执行在ELF文件中标记为 .init 的节的代码，进行程序运行的初始化。在早期系统中，初始化代码对应函数 \_init(void)(函数名强制固定)，在现代系统中，则对应形式为

|  |
| --- |
| void  \_\_attribute((constructor))  init\_function(void)  {  ……  } |

其中函数名为任意。

8：动态连接器把控制传递给程序，从 ELF 文件头部中定义的程序进入点开始执行。**在 a.out 格式和ELF格式中，程序进入点的值是显式存在的，**在 COFF 格式中则是由规范隐含定义。

从上面的描述可以看出，加载文件最重要的是完成两件事情：加载程序段和数据段到内存；进行外部定义符号的重定位。重定位是程序连接中一个重要概念。我们知道，一个可执行程序通常是由一个含有 main() 的主程序文件、若干目标文件、若干共享库（Shared Libraries）组成。（注：采用一些特别的技巧，也可编写没有 main 函数的程序，请参阅 [参考资料 2](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)）一个 C 程序可能引用共享库定义的变量或函数，换句话说就是程序运行时必须知道这些变量/函数的地址。在静态连接中，程序所有需要使用的外部定义都完全包含在可执行程序中，而动态连接则只在可执行文件中设置相关外部定义的一些引用信息，真正的重定位是在程序运行之时。静态连接方式有两个大问题：如果库中变量或函数有任何变化都必须重新编译连接程序；如果多个程序引用同样的变量/函数，则此变量/函数会在文件/内存中出现多次，浪费硬盘/内存空间。比较两种连接方式生成的可执行文件的大小，可以看出有明显的区别。

|  |
| --- |
| http://www.ibm.com/i/v14/rules/blue_rule.gif http://www.ibm.com/i/c.gif |

|  |  |  |
| --- | --- | --- |
| http://www.ibm.com/i/c.gif   |  |  | | --- | --- | | http://www.ibm.com/i/v14/icons/u_bold.gif | [**回页首**](http://www.ibm.com/developerworks/cn/linux/l-excutff/#main) | |

**a.out 文件格式分析**

a.out 格式在不同的机器平台和不同的 UNIX 操作系统上有轻微的不同，例如在 MC680x0 平台上有 6 个 section。下面我们讨论的是最"标准"的格式。

a.out 文件包含 7 个 section，格式如下：

|  |
| --- |
| exec header（执行头部，也可理解为文件头部） |
| text segment（文本段） |
| data segment(数据段) |
| text relocations(文本重定位段) |
| data relocations（数据重定位段） |
| symbol table（符号表） |
| string table（字符串表） |

执行头部的数据结构：

|  |
| --- |
| struct exec {  unsigned long a\_midmag; /\* 魔数和其它信息 \*/  unsigned long a\_text; /\* 文本段的长度 \*/  unsigned long a\_data; /\* 数据段的长度 \*/  unsigned long a\_bss; /\* BSS段的长度 \*/  unsigned long a\_syms; /\* 符号表的长度 \*/  unsigned long a\_entry; /\* 程序进入点 \*/  unsigned long a\_trsize; /\* 文本重定位表的长度 \*/  unsigned long a\_drsize; /\* 数据重定位表的长度 \*/  }; |

文件头部主要描述了各个 section 的长度，**比较重要的字段是 a\_entry（程序进入点），代表了系统在加载程序并初试化各种环境后开始执行程序代码的入口。**这个字段在后面讨论的 ELF 文件头部中也有出现。由 a.out 格式和头部数据结构我们可以看出，a.out 的格式非常紧凑，只包含了程序运行所必须的信息（文本、数据、BSS），而且每个 section 的顺序是固定的。这种结构缺乏扩展性，如不能包含"现代"可执行文件中常见的调试信息，最初的 UNIX 黑客对 a.out 文件调试使用的工具是 adb，而 adb 是一种机器语言调试器！

**a.out 文件中包含符号表和两个重定位表**，这三个表的内容在连接目标文件以生成可执行文件时起作用。在最终可执行的 a.out 文件中，这三个表的长度都为 0。**a.out 文件在连接时就把所有外部定义包含在可执行程序中，**如果从程序设计的角度来看，这是一种硬编码方式，或者可称为模块之间是强藕和的。在后面的讨论中，我们将会具体看到ELF格式和动态连接机制是如何对此进行改进的。

a.out 是早期UNIX系统使用的可执行文件格式，由 AT&T 设计，现在基本上已被 ELF 文件格式代替。a.out 的设计比较简单，但其设计思想明显的被后续的可执行文件格式所继承和发扬。可以参阅 [参考资料 16](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources) 和阅读 [参考资料 15](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources) 源代码加深对 a.out 格式的理解。 [参考资料 12](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources) 讨论了如何在"现代"的红帽LINUX运行 a.out 格式文件。

|  |
| --- |
| http://www.ibm.com/i/v14/rules/blue_rule.gif http://www.ibm.com/i/c.gif |

|  |  |  |
| --- | --- | --- |
| http://www.ibm.com/i/c.gif   |  |  | | --- | --- | | http://www.ibm.com/i/v14/icons/u_bold.gif | [**回页首**](http://www.ibm.com/developerworks/cn/linux/l-excutff/#main) | |

**COFF 文件格式分析**

COFF 格式比 a.out 格式要复杂一些，最重要的是包含一个节段表(section table)，因此除了 .text，.data，和 .bss 区段以外，还可以包含其它的区段。另外也多了一个可选的头部，不同的操作系统可一对此头部做特定的定义。

COFF 文件格式如下：

|  |
| --- |
| File Header(文件头部) |
| Optional Header(可选文件头部) |
| Section 1 Header(节头部) |
| ……… |
| Section n Header(节头部) |
| Raw Data for Section 1(节数据) |
| Raw Data for Section n(节数据) |
| Relocation Info for Sect. 1(节重定位数据) |
| Relocation Info for Sect. n(节重定位数据) |
| Line Numbers for Sect. 1(节行号数据) |
| Line Numbers for Sect. n(节行号数据) |
| Symbol table(符号表) |
| String table(字符串表) |

文件头部的数据结构：

|  |
| --- |
| struct filehdr  {  unsigned short f\_magic; /\* 魔数 \*/  unsigned short f\_nscns; /\* 节个数 \*/  long f\_timdat; /\* 文件建立时间 \*/  long f\_symptr; /\* 符号表相对文件的偏移量 \*/  long f\_nsyms; /\* 符号表条目个数 \*/  unsigned short f\_opthdr; /\* 可选头部长度 \*/  unsigned short f\_flags; /\* 标志 \*/  }; |

COFF 文件头部中魔数与其它两种格式的意义不太一样，它是表示针对的机器类型，例如 0x014c 相对于 I386 平台，而 0x268 相对于 Motorola 68000系列等。当 COFF 文件为可执行文件时，字段 f\_flags 的值为 F\_EXEC（0X00002），同时也表示此文件没有未解析的符号，换句话说，也就是重定位在连接时就已经完成。由此也可以看出，原始的 COFF 格式不支持动态连接。为了解决这个问题以及增加一些新的特性，一些操作系统对 COFF 格式进行了扩展。Microsoft 设计了名为 PE（Portable Executable）的文件格式，主要扩展是在 COFF 文件头部之上增加了一些专用头部，具体细节请参阅 [参考资料 18](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)，某些 UNIX 系统也对 COFF 格式进行了扩展，如 XCOFF（extended common object file format）格式，支持动态连接，请参阅 [参考资料 5](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)。

紧接文件头部的是可选头部，COFF 文件格式规范中规定可选头部的长度可以为 0，但在 LINUX 系统下可选头部是必须存在的。下面是 LINUX 下可选头部的数据结构：

|  |
| --- |
| typedef struct  {  char magic[2]; /\* 魔数 \*/  char vstamp[2]; /\* 版本号 \*/  char tsize[4]; /\* 文本段长度 \*/  char dsize[4]; /\* 已初始化数据段长度 \*/  char bsize[4]; /\* 未初始化数据段长度 \*/  char entry[4]; /\* 程序进入点 \*/  char text\_start[4]; /\* 文本段基地址 \*/  char data\_start[4]; /\* 数据段基地址 \*/  }  COFF\_AOUTHDR; |

字段 magic 为 0413 时表示 COFF 文件是可执行的，注意到可选头部中显式定义了程序进入点，标准的 COFF 文件没有明确的定义程序进入点的值，通常是从 .text 节开始执行，但这种设计并不好。

前面我们提到，COFF 格式比 a.out 格式多了一个节段表，一个节头条目描述一个节数据的细节，因此 COFF 格式能包含更多的节，或者说可以根据实际需要，增加特定的节，具体表现在 COFF 格式本身的定义以及稍早提及的 COFF 格式扩展。我个人认为，节段表的出现可能是 COFF 格式相对 a.out 格式最大的进步。下面我们将简单描述 COFF 文件中节的数据结构，因为节的意义更多体现在程序的编译和连接上，所以本文不对其做更多的描述。此外，ELF 格式和 COFF格式对节的定义非常相似，在随后的 ELF 格式分析中，我们将省略相关讨论。

|  |
| --- |
| struct COFF\_scnhdr  {  char s\_name[8]; /\* 节名称 \*/  char s\_paddr[4]; /\* 物理地址 \*/  char s\_vaddr[4]; /\* 虚拟地址 \*/  char s\_size[4]; /\* 节长度 \*/  char s\_scnptr[4]; /\* 节数据相对文件的偏移量 \*/  char s\_relptr[4]; /\* 节重定位信息偏移量 \*/  char s\_lnnoptr[4]; /\* 节行信息偏移量 \*/  char s\_nreloc[2]; /\* 节重定位条目数 \*/  char s\_nlnno[2]; /\* 节行信息条目数 \*/  char s\_flags[4]; /\* 段标记 \*/  }; |

有一点需要注意：LINUX系统中头文件coff.h中对字段s\_paddr的注释是"physical address"，但似乎应该理解为"节被加载到内存中所占用的空间长度"。字段s\_flags标记该节的类型，如文本段、数据段、BSS段等。在COFF的节中也出现了行信息，行信息描述了二进制代码与源代码的行号之间的对映关系，在调试时很有用。

[参考资料 19](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)是一份对COFF格式详细描述的中文资料，更详细的内容请参阅 [参考资料 20](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)。

|  |
| --- |
| http://www.ibm.com/i/v14/rules/blue_rule.gif http://www.ibm.com/i/c.gif |

|  |  |  |
| --- | --- | --- |
| http://www.ibm.com/i/c.gif   |  |  | | --- | --- | | http://www.ibm.com/i/v14/icons/u_bold.gif | [**回页首**](http://www.ibm.com/developerworks/cn/linux/l-excutff/#main) | |

**ELF文件格式分析**

ELF文件有三种类型： **可重定位文件：也就是通常称的目标文件，后缀为.o。** **共享文件：也就是通常称的库文件，后缀为.so。** **可执行文件**：本文主要讨论的文件格式，总的来说，可执行文件的格式与上述两种文件的格式之间的区别主要在于观察的角度不同：一种称为连接视图（Linking View），一种称为执行视图（Execution View）。

首先看看ELF文件的总体布局：

|  |
| --- |
| ELF header(ELF头部) |
| Program header table(程序头表) |
| Segment1（段1） |
| Segment2（段2） |
| ……… |
| Sengmentn（段n） |
| Setion header table(节头表，可选) |

段由若干个节(Section)构成,节头表对每一个节的信息有相关描述。对可执行程序而言，节头表是可选的。 [参考资料 1](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)中作者谈到把节头表的所有数据全部设置为0，程序也能正确运行！ELF头部是一个关于本文件的路线图（road map），从总体上描述文件的结构。下面是ELF头部的数据结构：

|  |
| --- |
| typedef struct  {  unsigned char e\_ident[EI\_NIDENT]; /\* 魔数和相关信息 \*/  Elf32\_Half e\_type; /\* 目标文件类型 \*/  Elf32\_Half e\_machine; /\* 硬件体系 \*/  Elf32\_Word e\_version; /\* 目标文件版本 \*/  Elf32\_Addr e\_entry; /\* 程序进入点 \*/  Elf32\_Off e\_phoff; /\* 程序头部偏移量 \*/  Elf32\_Off e\_shoff; /\* 节头部偏移量 \*/  Elf32\_Word e\_flags; /\* 处理器特定标志 \*/  Elf32\_Half e\_ehsize; /\* ELF头部长度 \*/  Elf32\_Half e\_phentsize; /\* 程序头部中一个条目的长度 \*/  Elf32\_Half e\_phnum; /\* 程序头部条目个数 \*/  Elf32\_Half e\_shentsize; /\* 节头部中一个条目的长度 \*/  Elf32\_Half e\_shnum; /\* 节头部条目个数 \*/  Elf32\_Half e\_shstrndx; /\* 节头部字符表索引 \*/  } Elf32\_Ehdr; |

下面我们对ELF头表中一些重要的字段作出相关说明，完整的ELF定义请参阅 [参考资料6](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)和 [参考资料 7](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)。

e\_ident[0]-e\_ident[3]包含了ELF文件的魔数，依次是0x7f、'E'、'L'、'F'。注意，任何一个ELF文件必须包含此魔数。 [参考资料 3](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)中讨论了利用程序、工具、/Proc文件系统等多种查看ELF魔数的方法。e\_ident[4]表示硬件系统的位数，1代表32位，2代表64位。e\_ident[5]表示数据编码方式，1代表小印第安排序（最大有意义的字节占有最低的地址），2代表大印第安排序（最大有意义的字节占有最高的地址）。e\_ident[6]指定ELF头部的版本，当前必须为1。e\_ident[7]到e\_ident[14]是填充符，通常是0。ELF格式规范中定义这几个字节是被忽略的，但实际上是这几个字节完全可以可被利用。如病毒Lin/Glaurung.676/666（ [参考资料 1](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)）设置e\_ident[7]为0x21,表示本文件已被感染；或者存放可执行代码（ [参考资料 2](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)）。ELF头部中大多数字段都是对子头部数据的描述，其意义相对比较简单。值得注意的是某些病毒可能修改字段e\_entry（程序进入点）的值，以指向病毒代码，例如上面提到的病毒Lin/Glaurung.676/666。

一个实际可执行文件的文件头部形式如下：（利用命令readelf）

|  |
| --- |
| ELF Header:  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  Class: ELF32  Data: 2's complement, little endian  Version: 1 (current)  OS/ABI: UNIX - System V  ABI Version: 0  Type: EXEC (Executable file)  Machine: Intel 80386  Version: 0x1  Entry point address: 0x80483cc  Start of program headers: 52 (bytes into file)  Start of section headers: 14936 (bytes into file)  Flags: 0x0  Size of this header: 52 (bytes)  Size of program headers: 32 (bytes)  Number of program headers: 6  Size of section headers: 40 (bytes)  Number of section headers: 34  Section header string table index: 31 |

紧接ELF头部的是程序头表，它是一个结构数组，包含了ELF头表中字段e\_phnum定义的条目，结构描述一个段或其他系统准备执行该程序所需要的信息。

|  |
| --- |
| typedef struct {  Elf32\_Word p\_type; /\* 段类型 \*/  Elf32\_Off p\_offset; /\* 段位置相对于文件开始处的偏移量 \*/  Elf32\_Addr p\_vaddr; /\* 段在内存中的地址 \*/  Elf32\_Addr p\_paddr; /\* 段的物理地址 \*/  Elf32\_Word p\_filesz; /\* 段在文件中的长度 \*/  Elf32\_Word p\_memsz; /\* 段在内存中的长度 \*/  Elf32\_Word p\_flags; /\* 段的标记 \*/  Elf32\_Word p\_align; /\* 段在内存中对齐标记 \*/  } Elf32\_Phdr; |

在详细讨论可执行文件程序头表之前，首先查看一个实际文件的输出：

|  |
| --- |
| Program Headers:  Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align  PHDR 0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4  INTERP 0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R 0x1  [Requesting program interpreter: /lib/ld-linux.so.2]  LOAD 0x000000 0x08048000 0x08048000 0x00684 0x00684 R E 0x1000  LOAD 0x000684 0x08049684 0x08049684 0x00118 0x00130 RW 0x1000  DYNAMIC 0x000690 0x08049690 0x08049690 0x000c8 0x000c8 RW 0x4  NOTE 0x000108 0x08048108 0x08048108 0x00020 0x00020 R 0x4  Section to Segment mapping:  Segment Sections...  00  01 .interp  02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version\_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh\_frame  03 .data .dynamic .ctors .dtors .jcr .got .bss  04 .dynamic  05 .note.ABI-tag  Section Headers:  [Nr] Name Type Addr Off Size ES Flg Lk Inf Al  [ 0] NULL 00000000 000000 000000 00 0 0 0  [ 1] .interp PROGBITS 080480f4 0000f4 000013 00 A 0 0 1  [ 2] .note.ABI-tag NOTE 08048108 000108 000020 00 A 0 0 4  [ 3] .hash HASH 08048128 000128 000040 04 A 4 0 4  [ 4] .dynsym DYNSYM 08048168 000168 0000b0 10 A 5 1 4  [ 5] .dynstr STRTAB 08048218 000218 00007b 00 A 0 0 1  [ 6] .gnu.version VERSYM 08048294 000294 000016 02 A 4 0 2  [ 7] .gnu.version\_r VERNEED 080482ac 0002ac 000030 00 A 5 1 4  [ 8] .rel.dyn REL 080482dc 0002dc 000008 08 A 4 0 4  [ 9] .rel.plt REL 080482e4 0002e4 000040 08 A 4 b 4  [10] .init PROGBITS 08048324 000324 000017 00 AX 0 0 4  [11] .plt PROGBITS 0804833c 00033c 000090 04 AX 0 0 4  [12] .text PROGBITS 080483cc 0003cc 0001f8 00 AX 0 0 4  [13] .fini PROGBITS 080485c4 0005c4 00001b 00 AX 0 0 4  [14] .rodata PROGBITS 080485e0 0005e0 00009f 00 A 0 0 32  [15] .eh\_frame PROGBITS 08048680 000680 000004 00 A 0 0 4  [16] .data PROGBITS 08049684 000684 00000c 00 WA 0 0 4  [17] .dynamic DYNAMIC 08049690 000690 0000c8 08 WA 5 0 4  [18] .ctors PROGBITS 08049758 000758 000008 00 WA 0 0 4  [19] .dtors PROGBITS 08049760 000760 000008 00 WA 0 0 4  [20] .jcr PROGBITS 08049768 000768 000004 00 WA 0 0 4  **[21] .got PROGBITS 0804976c 00076c 000030 04 WA 0 0 4**  [22] .bss NOBITS 0804979c 00079c 000018 00 WA 0 0 4  [23] .comment PROGBITS 00000000 00079c 000132 00 0 0 1  [24] .debug\_aranges PROGBITS 00000000 0008d0 000098 00 0 0 8  [25] .debug\_pubnames PROGBITS 00000000 000968 000040 00 0 0 1  [26] .debug\_info PROGBITS 00000000 0009a8 001cc6 00 0 0 1  [27] .debug\_abbrev PROGBITS 00000000 00266e 0002cc 00 0 0 1  [28] .debug\_line PROGBITS 00000000 00293a 0003dc 00 0 0 1  [29] .debug\_frame PROGBITS 00000000 002d18 000048 00 0 0 4  [30] .debug\_str PROGBITS 00000000 002d60 000bcd 01 MS 0 0 1  [31] .shstrtab STRTAB 00000000 00392d 00012b 00 0 0 1  [32] .symtab SYMTAB 00000000 003fa8 000740 10 33 56 4  [33] .strtab STRTAB 00000000 0046e8 000467 00 0 0 1 |

对一个ELF可执行程序而言，**一个基本的段是标记p\_type为PT\_INTERP的段，它表明了运行此程序所需要的程序解释器（/lib/ld-linux.so.2）**，实际上也就是动态连接器（dynamic linker）。**最重要的段是标记p\_type为PT\_LOAD的段**，它表明了为运行程序而需要加载到内存的数据。查看上面实际输入，可以看见有两个可LOAD段，第一个为只读可执行（FLg为RE）,第二个为可读可写（Flg为RW）。段1包含了文本节.text，注意到ELF文件头部中程序进入点的值为0x80483cc，正好是指向节.text在内存中的地址。段二包含了数据节.data，此数据节中数据是可读可写的，相对的只读数据节.rodata包含在段1中。ELF格式可以比COFF格式包含更多的调试信息，如上面所列出的形式为.debug\_xxx的节。在I386平台LINUX系统下，用命令file查看一个ELF可执行程序的可能输出是：a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped。

ELF文件中包含了动态连接器的全路径，**内核定位"正确"的动态连接器在内存中的地址是"正确"运行可执行文件的保证，**[**参考资料 13**](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)**讨论了如何通过查找动态连接器在内存中的地址以达到颠覆（Subversiver）动态连接机制的方法。**

最后我们讨论ELF文件的动态连接机制。**每一个外部定义的符号在全局偏移表(Global Offset Table GOT)中有相应的条目,如果符号是函数则在过程连接表(Procedure Linkage Table PLT)中也有相应的条目**，且一个PLT条目对应一个GOT条目。**对外部定义函数解析可能是整个ELF文件规范中最复杂的，**下面是函数符号解析过程的一个描述。

1：代码中调用外部函数func,语句形式为call 0xaabbccdd,地址0xaabbccdd实际上就是符号func在PLT表中对应的条目地址（假设地址为标号.PLT2）。

2：PLT表的形式如下

|  |
| --- |
| **.PLT0: pushl 4(%ebx) /\* GOT表的地址保存在寄存器ebx中 \*/**  jmp \*8(%ebx)  nop; nop  nop; nop  .PLT1: jmp \*name1@GOT(%ebx)  pushl $offset  jmp .PLT0@PC  .PLT2: jmp \*func@GOT(%ebx)  pushl $offset  jmp .PLT0@PC |

3：查看标号.PLT2的语句,实际上是跳转到符号func在GOT表中对应的条目。

4：在符号没有重定位前，GOT表中此符号对应的地址为标号.PLT2的下一条语句，即是pushl $offset，其中$offset是符号func的重定位偏移量。注意到这是一个二次跳转。

5：在符号func的重定位偏移量压栈后,控制跳到PLT表的第一条目，把GOT[1]的内容压栈，并跳转到GOT[2]对应的地址。

6：GOT[2]对应的实际上是动态符号解析函数的代码，在对符号func的地址解析后，会把func在内存中的地址设置到GOT表中此符号对应的条目中。

7：当第二次调用此符号时，GOT表中对应的条目已经包含了此符号的地址，就可直接调用而不需要利用PLT表进行跳转。

动态连接是比较复杂的，但为了获得灵活性的代价通常就是复杂性。**其最终目的是把GOT表中条目的值修改为符号的真实地址，这也可解释节.got包含在可读可写段中。**

动态连接是一个非常重要的进步，这意味着库文件可以被升级、移动到其他目录等等而不需要重新编译程序（当然，这不意味库可以任意修改，如函数入参的个数、数据类型应保持兼容性）。从很大程度上说，动态连接机制是ELF格式代替a.out格式的决定性原因。如果说面对对象的编程本质是面对接口（interface）的编程，那么动态连接机制则是这种思想的地一个非常典型的应用，具体的讲，动态连接机制与设计模式中的桥接（BRIDGE）方法比较类似，而它的LAZY特性则与代理（PROXY）方法非常相似。动态连接操作的细节描述请参阅 [参考资料 8，9，10，11](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)。通过阅读命令readelf、objdump 的源代码以及 [参考资料 14](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)中所提及的相关软件源代码，可以对ELF文件的格式有更彻底的了解。

|  |
| --- |
| http://www.ibm.com/i/v14/rules/blue_rule.gif http://www.ibm.com/i/c.gif |

|  |  |  |
| --- | --- | --- |
| http://www.ibm.com/i/c.gif   |  |  | | --- | --- | | http://www.ibm.com/i/v14/icons/u_bold.gif | [**回页首**](http://www.ibm.com/developerworks/cn/linux/l-excutff/#main) | |

**总结**

不同时期的可执行文件格式深刻的反映了技术进步的过程，技术进步通常是针对解决存在的问题和适应新的环境。早期的UNIX系统使用a.out格式，随着操作系统和硬件系统的进步，a.out格式的局限性越来越明显。新的可执行文件格式COFF在UNIX System VR3中出现，COFF格式相对a.out格式最大变化是多了一个节头表（section head table），能够在包含基础的文本段、数据段、BSS段之外包含更多的段，但是COFF对动态连接和C++程序的支持仍然比较困难。为了解决上述问题，UNIX系统实验室(UNIX SYSTEM Laboratories USL) 开发出ELF文件格式，它被作为应用程序二进制接口（Application binary Interface ABI）的一部分，其目的是替代传统的a.out格式。例如，ELF文件格式中引入初始化段.init和结束段.fini（分别对应构造函数和析构函数）则主要是为了支持C++程序。1994年6月ELF格式出现在LINUX系统上，现在ELF格式作为UNIX/LINUX最主要的可执行文件格式。当然我们完全有理由相信，在将来还会有新的可执行文件格式出现。

上述三种可执行文件格式都很好的体现了设计思想中分层的概念，由一个总的头部刻画了文件的基本要素，再由若干子头部/条目刻画了文件的若干细节。比较一下可执行文件格式和以太数据包中以太头、IP头、TCP头的设计，我想我们能很好的感受分层这一重要的设计思想。 [参考资料 21](http://www.ibm.com/developerworks/cn/linux/l-excutff/#resources)从全局的角度讨论了各种文件的格式，并提出一个比较夸张的结论：Everything Is Byte!

最后的题外话：大多数资料中对a.out格式的评价较低，常见的词语有黑暗年代（dark ages）、丑陋（ugly）等等，当然，从现代的观点来看，的确是比较简单，但是如果没有曾经的简单何来今天的精巧？正如我们今天可以评价石器时代的技术是ugly,那么将来的人们也可以嘲讽今天的技术是非常ugly。我想我们也许应该用更平和的心态来对曾经的技术有一个公正的评价。

**参考资料**

1. 《 [A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux](http://www.xfocus.net/articles/200103/107.html) 》 breadbox
2. 《 [The Linux Virus Writing And Detection HOWTO](http://www.madchat.org/vxdevl/papers/%20vxers/nux/writingvirusinlinux.pdf)》Alexander Bartolich
3. 《 [从程序员角度看ELF](http://www.xfocus.net/articles/200109/260.html)》Hongjiu Lu alert7（译）
4. 《 [XCOFF Object File Format](http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/files/aixfiles/XCOFF.htm)》
5. 《 [Executable and Linkable Format(ELF)](http://www.cs.princeton.edu/courses/%20archive/fall04/cos217/reading/elf.pdf)》
6. 《 [如何修改动态库符号表](http://elfhack.whitecell.org/chinesedocs/howto_modity_symbol.txt)》wangdb
7. 《 [分析ELF的加载过程](http://www.linuxforum.net/forum/showflat.php?Cat=&Board=linuxK&Number=102254&page=162&view=collapsed&sb=5&o=all)》opera
8. 《 [Before main() 分析](http://www.xfocus.net/articles/200109/269.html)》 alert7
9. 《 [Linkers & Loaders](http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=program&Number=483117&page=0&view=collapsed&sb=5&o=all&fpart=)》John R. Levine
10. 《 [Running a.out executables on modern Red Hat Linux](http://www.ofb.net/~jheiss/aout_redhat.shtml)》
11. 《 [Cheating the ELF](http://alge.anart.no/projects/free_books/Subversiveld.pdf)》
12. 《 [Manual Reference Pages - A.OUT (5)](http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=a.out)》
13. 《 [Linux下缓冲区溢出攻击的原理及对策](http://blog.csdn.net/lalphbet/archive/2004/11/14/181343.aspx)》
14. 《 [Microsoft Portable Executable and Common Object File Format Specification](http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx)》
15. 《 [COFF的文件结构](http://dev.csdn.net/user/redleaves)》redleaves
16. 《 [Everything Is Byte](http://www.reteam.org/papers/e56.pdf)》 mala

**关于作者**