二进制的☆秘☆密☆

Published at Draft [1] 2014-06-05 02:07 Wordage: 3442 1686 views

如何用命令行工具审查二进制文件

相像一下你手头有一个可执行的二进制文件，只要你运行它电脑就会立刻爆炸。聪明的你马上调用 `chmod -x` 来防止这个情况发生。但是你仍然很好奇，想知道这个程序的各种信息，比如它是否是 64 位的，用什么编译器编译的，依赖于什么库等等。这时你应该怎么做呢？

Linux 下有一系列的工具都是用来做这些事，让你在不执行一个程序的情况下查看其各种信息。

相关基础知识

在开始之前我们需要了解的是一些相关的知识。虽然说很基础但是我其实也是真的要用到的时候才去看了下。

首先是二进制文件的格式。显然的可执行文件必须满足某种格式才能真的被系统读取并执行。目前常见 *nix 系统都是使用的 ELF[2] 格式。Windows 使用的是 PE(Portable Executable)[3] 格式，是在另一个古老的格式 COFF[4] 上改进而来的。

举一个例子，对于内容如下的一个文件[5]：

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |........T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |........4. .....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
00000040  00 80 04 08 6f 00 00 00  6f 00 00 00 05 00 00 00  |....o...o.......|
00000050  00 10 00 00 b0 04 b2 0c  b9 63 80 04 08 cd 80 b0  |.........c......|
00000060  01 cd 80 48 65 6c 6c 6f  20 77 6f 72 6c 64 0a     |...Hello world.|
```

就是一个合法的 ELF 格式文件。在 Linux 下此文件如果被设置为可执行的话，通过 Shell 执行时 Linux 就会先根据前面的 `.ELF` 头来判断这是一个 ELF 文件，然后使用再又 Loader 验证并载入内存最后开始执行。本文后面大部分内容都是针对 ELF 格式来讲的。

在之后是关于 Linker 的东西。网上有一份 `gold` 作者写的一系列文章[6]非常详细的介绍了 Linker 这个熟悉而又陌生的东西到底做了些什么。说实话其内容有点太过高深，看过以后我就记得这几个东西的东西：

- soname[7], 也就是动态链接库 ("libfoo.so" 这类) 的名字和它相关的几个东西。比如你编译的时候用是写的 `gcc bar.c -lfoo -o bar`，那么你 link 的是 `libfoo`，对应的 **link name** 是 `libfoo.so`。但事实上记录到你编译出来的 `bar` 文件中的可能是 `libfoo.so.2`，对应了一个具体的版本，这个名字就是 **soname**。而事实上当程序运行起来的时候它找到的动态链接库文件可能是 `libfoo.bar.2.1`，这个被称作 **fully qualified name**。其中 `soname` 中的版本号对应着相互可以不兼容的 `libfoo` 的不同版本，而 **fully qualified name** 中最后一个小版本号记录着这是第几个 "fix"。
- RPATH 和 RUNPATH 是 ELF 文件中记录 “到哪里去找各种 .so 文件” 的东西。你知道 Windows 上的 dll 文件会现在当前工作目录下找，然后在 `PATH` 上一个个找。在 Linux 下运行时查找 .so 文件的方法除了根据系统中的一些预设值和规则，还会沿着 `RPATH` 和 `RUNPATH` 一个个找。两者之间的区别貌似是在查找的优先级上。
- 动态链接要在程序运行时做各种神奇的事情，那么完成这些东西的代码其实都是在 `ld.so` 里面。动态链接的程序并不是这真正完整的，在运行时需要 `ld.so` 来做很多事情。一个神奇的例子是你的程序中首次调用任何一个外部动态库中的函数的时候都会有额外开销，`ld.so` 会通过 ELF 文件中的信息来找到这个函数的真正内存地址并把它记录在叫做 Procedure Linkage Table(PLT) 的表中，使得后续的调用可以省去这一步。Linux 下可以用 `man ld.so` 查看相关文档。

我在自己的机器上编译一个简单的 C 文件：

```
$ cat > bar.c
#include <stdio.h>

void foo() {
    puts("hello world.");
}

int main() {
    foo();
```

```
    return 0;
}
$ gcc -g bar.c -o bar
$ ./bar
hello world.
```

后文会用到产生的二进制文件 `bar` 作为例子。

一上来当然应该用万能的 `file` 来看看它是啥：

```
$ file bar
bar: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=0
```

`file` 的工作原理是根据系统中的 `magic` 文件，读取 `bar` 开头的部分来判定文件的类型。之后对于某些文件类型可以继续的读取相关信息并解析出来。这里 `file` 认出了 `bar` 是一个 ELF 格式的可执行文件，并从 ELF 头来读出了关于大小端，32/64架构，是否动态链接等等信息。末尾的 `not stripped` 指的是文件包含调试信息。

要看到像文章开头那样高端的十六进制加 ASCII 码表示，我们可以使用 `xxd`:

```
$ xxd bar | head -4
0000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
0000010: 0200 3e00 0100 0000 3004 4000 0000 0000  ..>.....0.@.....
0000020: 4000 0000 0000 0000 880d 0000 0000 0000  @...............
0000030: 0000 0000 4000 3800 0800 4000 2500 2200  ....@.8...@.%.".
```

可以清楚的看到最开始四位中的 `ELF`，这也是 ELF 格式故意定义的标识。

用 `strings` 可以看到文件中的 C 风格字符串：

```
$ strings bar | tail -4
UH-H    `
UH-H    `
[]A\A]A^A_
hello world.
```

这里很清楚的就能看到程序中的 "hello world."，可以想象程序中的字符串常量是多么的脆弱。通过上面的 `xxd` 我们还可以轻易的修改它：

```
$ echo "0005da: 6865 6c6c 6f20" | xxd -r - bar
$ ./bar
hello hello
```

通过 `xxd -r` 我们把 "world." 对应的数据换成了 "hello "，重新执行程序输出也发生了相应的改变。

编译，链接和调试相关

用 `nm` 可以列出 ELF 中包含的 "Symbols"，比较明显的就是其自己内部的 C 函数名，以及用到外部的其他函数。由于我们编译时特地保留了调试信息，用 `nm -l` 可以看到 Symbol 对应的到哪个文件的哪一行：

```
$ nm -l bar | tail -5
000000000040051c T foo  /tmp/bar.c:3
00000000004004f0 t frame_dummy
000000000040052c T main /tmp/bar.c:7
                 U puts@@GLIBC_2.2.5
0000000000400490 t register_tm_clones
```

上面显示的三栏分别是 Symbol 的地址，类型和名字加上附加信息。类型中 `T` 代表符号在 ELF 的 `.text` section 中，也就是说这个符号的代码包含在该文件中。而 "U" 是说该符号还未定义，也就是需要运行时由 `ld.so` 链接并满足。

`bar` 甚至于绝大多数常见 Linux 可执行文件都是动态链接的。`ldd` 可以用来查看其依赖的动态链接库，也就是要运行 `bar` 所需要的其他 `.so` 文件们：

```
$ ldd bar
    linux-vdso.so.1 (0x00007fffecf1b000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0e6e123000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f0e6e4e1000)
```

其中 `libc.so.6` 就是上面提到的 `soname`。`libc` 中的内容是 C 标准库的东西。如同这篇文章里提到的，"C 语言最大的成就之一就是让大家认为 C 是

不需要 Runtime 的"，事实上绝大部分程序都跑不开对 libc 的依赖。最下面的 `ld-linux-x86-64.so.2` 就是之前提到的 `ld.so`，显然的不同架构使用的 `ld.so` 也是分开的。最开始的 `linux-vdso.so.1` 说实话我也是第一次注意到... 但根据文档来看[8]似乎是不太需要普通用户关心的东西。

比如我把 `bar` 拷贝到另一台系统目录结构有些差异的机器上，那么某些库可能找不到。`ldd` 在这种时候也能给出比较清楚的解释。另外 Windows 上也有一个类似的工具叫做 depends.exe[9] 用来查找 dll 依赖。

要确定 `bar` 到底是用哪个版本的编译器编译出来的，简单粗暴的方法就是使用 `strings -a` 来读取包括非用户区的所有字串符：

```
$ strings -a bar | grep GCC
GCC: (Debian 4.7.2-5) 4.7.2
GCC: (Debian 4.8.2-21) 4.8.2
```

事实上这里用的的确是 4.7.2，我也说不清后面的 4.8 是从哪里来的。要注意的是这些信息基本上是只在 ELF 文件带有调试信息的时候才会保留下来。

调试信息当然也是直接存在 ELF 文件中的。ELF 用不同的 **section** 来存放不同类型的数据，在执行的时候将 **section** 拼成 **segment** 来载入到内存。这里的 **section** 值的就是 `.text`, `.data` 这样的数据段。而 **segment** 正是我们熟悉的 `Segment Fault` 里面的那个 **segment**。我们可以用工具读取 ELF 文件并解析其内容。这里可以使用 `readelf` 和 `objdump`，两者功能上似乎没有太大的区别，本文后面用 `readelf` 作为示范。

用 `readelf -l` 列出所有的 **segment** 和其包含的 **section** 之间的关系。由于显示结果比较长，这里只列出裁减过的一部分：

```
$ readelf -l bar
Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
                 ..........................
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000704 0x0000000000000704  R E    200000
  LOAD           0x0000000000000708 0x0000000000600708 0x0000000000600708
                 0x0000000000000240 0x0000000000000248  RW     200000
                 ..........................

 Section to Segment mapping:
  Segment Sections...
                 ..........................
   02     .plt .text .fini .rodata
   03     .data .bss
                 ..........................
```

首先来看看 **section**，这个估计大家都有一点印象：

- `.text` 存放的是编译过后的代码，也就是可以执行的机器指令。
- `.rodata` 存放的是只读的数据，比如像我们程序中的 "hello world" 常量字符串。
- `.data` 存放的是可读写的数据，比如 `int foo` 这样的全局变量。
- `.bss` 存放的是未初始化的数据段。

对应的我们有两个 **section** 分别是 `02` 和 `03`，两者的最大差别在于 `FLAGS` 那一栏。这里的 `FLAGS` 跟 Linux 文件系统的权限是一个意思，`RWE` 分别对应可读，可写和可行。`02` 是可读可执行不能写的，所以 `.text`, `.rodata` 两个 **section** 被分配到了这个 **segment**。系统在载入 `02` 时会根据 FLAGS 对其进行保护，如果你数组越界写到了 `02` 这个不可写的 **segment** 上，就会产生 `Segment Fault`。同理因为 `.data` 和 `.bss` 都是需要可以被写的，所以他们被分配到了 `03` 上。显然如果 `03` 中的数据能被当做指令执行会异常危险，所以系统也会保证如果 PC[10] 指到这边的话也会报错。

`readelf` 也可以用来查看某个 **section** 中的内容。很容易想象 `.rodata` 中应该有 "hello world." 这个字符串，这里用 `readelf -x .rodata` 来确认一下：

```
$ readelf -x .rodata bar

Hex dump of section '.rodata':
  0x004005d0 01000200 68656c6c 6f20776f 726c642e ....hello world.
  0x004005e0 00                                  .
```

事实上不是所有的 **section** 都会被分到 **segment** 并载入内存。用 `readelf -S` 可以列出所有的 **section**：

```
$ readelf -S bar
There are 37 section headers, starting at offset 0xd88:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
                  ..........................
  [12] .init             PROGBITS         00000000004003d0  000003d0
       000000000000001a  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         00000000004003f0  000003f0
       0000000000000040  0000000000000010  AX       0     0     16
  [14] .text             PROGBITS         0000000000400430  00000430
       0000000000000194  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         00000000004005c4  000005c4
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         00000000004005d0  000005d0
       0000000000000011  0000000000000000   A       0     0     4
                  ..........................
  [25] .data             PROGBITS         0000000000600938  00000938
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000600948  00000948
       0000000000000008  0000000000000000  WA       0     0     4
  [27] .comment          PROGBITS         0000000000000000  00000948
       0000000000000039  0000000000000001  MS       0     0     1
  [28] .debug_aranges    PROGBITS         0000000000000000  00000981
       0000000000000030  0000000000000000           0     0     1
  [29] .debug_info       PROGBITS         0000000000000000  000009b1
       00000000000000b2  0000000000000000           0     0     1
```

可以看到像 `.comment` 和 debug 相关的 **section** 明显没有被分到任何 **segment** 里。`.comment` 里面放的正是之前看到的编译器信息：

```
$ readelf -x .comment bar

Hex dump of section '.comment':
  0x00000000 4743433a 20284465 6269616e 20342e37 GCC: (Debian 4.7
  0x00000010 2e322d35 2920342e 372e3200 4743433a .2-5) 4.7.2.GCC:
  0x00000020 20284465 6269616e 20342e38 2e322d32  (Debian 4.8.2-2
  0x00000030 31292034 2e382e32 00                1) 4.8.2.
```

而各种 debug 相关的信息则需要 `gdb` 这样的工具来读取并解析：

```
$ gdb -batch -ex "info sources" bar
Source files for which symbols will be read in on demand:
/tmp/bar.c
```

可以看到 `gdb` 能够找到源文件的路径。

使用 `strip` 可以将调试信息从 ELF 中消去:

```
$ strip bar
$ readelf -S bar
There are 29 section headers, starting at offset 0xa80:
                  ..........................
```

可以看到 **section** 数量减少了很多。

不知道你有没有想到这些东西在实际工作中有什么用处。说实话这些可以用的地方不是太多... 但偶尔碰需要处理环境相关的情况，或者验证编译出的文件是否有问题，这些东西应该能派的上用场。

1. http://jianshu.io/notebooks/11259/latest
2. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
3. http://en.wikipedia.org/wiki/Portable_Executable
4. http://codegolf.stackexchange.com/a/5726
5. http://codegolf.stackexchange.com/a/5726
6. http://lwn.net/Articles/276782/
7. http://en.wikipedia.org/wiki/Soname
8. http://man7.org/linux/man-pages/man7/vdso.7.html
9. http://www.dependencywalker.com/
10. http://en.wikipedia.org/wiki/Program_counter