man ld.so 的翻译

原创 longyu_wlz 于 2020-09-10 12:50:45 发布 219 收藏 版权

分类专栏： Linux 文章标签： ld.so 动态库链接器 ld-linux-so 动态库链接器的使用 动态库相关环境变量

动态库加载器可以通过运行一些 动态链接 的程序来间接调用（在这种情况中，不需要向动态库加载器指定任何命令行参数，对于 ELF 文件，动态库链接器的路径存储在将被执行的程序的 .interp section 中）或直接通过 /lib/ld-linux-so.* xx 来调用。

ld.so 与 ld-linux.so* 查找并加载程序依赖的动态库，为程序正常运行准备动态库环境，然后再运行程序。

Linux 中的 二进制 程序默认都是动态链接的，除非在编译时向 ld 命令指定 -static 选项。

ld.so 处理 a.out 格式的二进制，ld-linux.so*（libc5 中为 /lib/ld-linux.so.1，glibc2 中为 /lib/ld-linux.so.2 ） 处理 ELF 格式（一种每个人都使用了很多年的文件格式）的文件。

这两个加载器有相同的行为，他们都支持文件与 ldd 程序、ldconfig 与 /etc/ld.so.conf.

程序依赖的共享库按照如下路径顺序进行查找：

1. 针对 ELF 格式文件，当 DT_RUNPATH 属性不存在的情况下，使用二进制程序 dynamic section 中存在的 DT_RPATH 属性指定的路径来搜索 。DT_RPATH 已经被弃用。
2. 使用环境变量 LD_LIBRARY_PATH 中指定的路径来搜索。如果可执行程序设定了 setuid/setgid，这一步将被跳过。
3. 从缓存文件 /etc/ld.so.cache 中查找。如果程序在链接时使用了 -z nodeflib 选项，默认库路径中的库及那个会被跳过。安装到硬件兼容目录中的库将会比其它库优先查找。
4. 在默认的 /lib 然后时 /usr/lib 中寻找，如果程序在链接时使用了 -z nodeflib 选项，这一步将被跳过。

$ORIGIN and rpath

ld.so 将在 rpath 中指定的 $ORIGIN 字符串作为包含可执行程序的目录。因此一个位于 somedir/app 的程序可以使用 `gcc -Wl,-rpath,'$ORIGIN/../lib'` 来编译，这样他竟会在 somedir/lib 路径中来查找相关的动态库而不用关心 somedir 在目录层次中的位置。

这方便了创建一个可以“交钥匙”的程序，这种程序不需要被安装到特定的目录中，可以被解压到任一目录中并且仍然能够找到它们自己的共享库。

```
OPTIONS

        --list 列出所有的依赖，和解决依赖的结果

        --verify 验证程序是动态链接的并且该动态库链接器能够处理它

        --library-path PATH 使用 PATH 代替 LD_LIBRARY_PATH 环境变量的设定。

        --inhibit-rpath LIST
             忽略目标对象 LIST 中的 RPATH 和 RUNPATH 信息，如果 ld.so 谁当了 setuid 或 setgid 这个参数将被忽略。

        --audit LIST
             使用目标文件中的 LIST 作为审核员。
```

HARDWARE CAPABILITIES

一些库使用特定的硬件指令编译，这些指令并不一定在每个 cpu 上都存在。这些库应该被安装路径名中包括需要的硬件特性的目录中，例如 /usr/lib/sse2.

当前不同架构能够识别的硬件特性名称如下:

```
     Alpha  ev4, ev5, ev56, ev6, ev67

       MIPS   loongson2e, loongson2f, octeon, octeon2

       PowerPC
              4xxmac,  altivec, arch_2_05, arch_2_06, booke, cellbe, dfp, efp-
              double, efpsingle,  fpu,  ic_snoop,  mmu,  notb,  pa6t,  power4,
              power5,  power5+,  power6x,  ppc32,  ppc601,  ppc64,  smt,  spe,
              ucache, vsx

       SPARC  flush, muldiv, stbar, swap, ultra3, v9, v9v, v9v2
```

```
    s390   dfp, eimm, esan3, etf3enh,  g5,  highgprs,  hpage,  ldisp,  msa,
           stfle, z900, z990, z9-109, z10, zarch


    x86 (32-bit only)
           acpi,  apic,  clflush,  cmov,  cx8,  dts, fxsr, ht, i386,
           i486, i586, i686, mca, mmx,  mtrr,  pat,  pbe,  pge,  pn,
           pse36, sep, ss, sse, sse2, tm
```

ENVIRONMENT

```
There are four important environment variables.

LD_BIND_NOW
       (libc5;  glibc  since 2.1.1) 当这个变量被设定为非空值时，动态库加载器将会在程序启动
       的时候解决所有依赖的符号，而不是在符号被第一次引用到的
       时才处理符号，这对于调试器非常有用。

LD_LIBRARY_PATH
       一个以冒号分隔的目录表，这个目录表在 ELF 程序执行时用来搜索动态库。与 PATH 环境变行为类似。

LD_PRELOAD
       一个以空白字符分割的额外的、用户指定的、需要在其它动态
       库加载前加载的动态库。

       设定这个变量能够选择性的覆盖其它动态库中的函数。对于 setuid、setgid 的 ELF
       程序，只有在标准搜索路径中也设定了 setuid
       的动态库才会被加载。

LD_TRACE_LOADED_OBJECTS

     只针对 ELF
     格式文件。当该变量被设定为非空值时，程序将会输出它依赖的
     动态库，像运行 ldd 一样的结果，而不会真正运行。

 除了这四个变量，还有一些内部使用的变量。我只列举几个有趣的变量。
```

```
26
27      LD_AUDIT
28
29           glibc 2.4 及之后的版本能够使用这个变量。能够设定审计点，例如在加载一个新的动态库、处理一个符号、调用一个符号时调用 audit 动态库重的函数来通知审计程序
30
31      LD_BIND_NOT
32             (glibc since 2.1.95) Do not update the GOT (global offset
33             table) and PLT (procedure linkage table) after  resolving
34             a symbol.
35
36      LD_DEBUG
37           glibc 2.1 之后的版本能够使用该变量。能够输出调试信息。
38
39      LD_DEBUG_OUTPUT
40
41           LD_DEBUG 开启后，输出信息输出位置，默认为 stdout.
42
43      LD_DYNAMIC_WEAK
44
45           glibc 2.1.91 及之后的版本。运行弱符号被覆盖。
46
47      LD_HWCAP_MASK
48             (glibc since 2.1) Mask for hardware capabilities..
49
50      LD_ORIGIN_PATH
51
52           glibc 2.1 及之后的版本。指定访问可执行程序的路径。
53
54      LD_POINTER_GUARD
55
56           gligc 2.4 之后。设定为 0 关闭 pointer guarding，其它值均表示开启 pointer guarding 功能，默认开启。
57
58           这是个安全机制，对于一些存储在可写程序内存的指针（setjmp
59           保存的返回地址、不同 glibc
60           内部使用的函数指针）以半随机方式分隔以提供攻击者利用缓冲
61           区溢出或堆栈溢出来劫持指针。
62
63
```

```
    LD_PROFILE

         profile 的动态库路径名，结果写入到

         "$LD_PROFILE_OUT-
           PUT/$LD_PROFILE.profile" 中。

    LD_PROFILE_OUTPUT

         glibc 2.1 及之后的版本。LD_PROFILE 输出的目录位置。默认值为 /var/tmp 目录

    LD_SHOW_AUXV
         glibc 2.1 输出从内内核传出的辅助性数组内容


    LD_USE_LOAD_BIAS
         默认情况下，可执行目标文件与预链接的动态库文件将指明它们
         依赖的动态库的基地址，（非预链接、位置无关的可执行程序）、其它共享库则不会指明。当
         LD_USER_LOAD_BIAS 变量被定义，可执行程序与
         PIE(位置无关的可执行部分) 也会指明这个基地址。如果 LD_USER_LOAD_BIAS 值为
         0，可执行文件与位置无关的可执行文件都不会指明这个及地址。

    LD_VERBOSE

         glibc 2.1
         如果该变量被设定为一个非空字符串，当请求检索程序的信息时
         (LD_TRACE_LOADED_OBJECTS 被设定或 --list 或 --verfiy
         选项传递给动态库链接器）将会输出符号的详细信息。

    LD_WARN
         只针对 ELF 格式文件。glibc 2.1.3 如果被设定为非空字符串，将会在未找到符号时报警。

    LDD_ARGV0

         libc5 中 ldd 使用的 argv[0] 的值，当没有一个可用时。
```

FILES

1. /lib/ld.so
   a.out 格式的动态库链接、加载器
2. /lib/ld-linux.so.{1,2}
   ELF 格式的动态库链接、加载器
3. /etc/ld.so.cache
   这个文件中包含了编译的动态库目录列表用于检索动态库以及一个排序的候选动态库列表。
4. /etc/ld.so.preload
   这个文件中包含了以空白字符分割的将在程序执行之前加载的动态库。
5. lib*.so*
   动态库

NOTES

ld.so 功能在使用 ligc 4.4.3 及之后的版本上可用。ELF 功能在 linux 1.1.52 与 libc5 及之后的版本都可使用。

LD_DEBUG 变量中的子命令：

```
 LD_DEBUG=help ls
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit
```

在解决动态库依赖问题时可能要用到。

对 setuid 与 setgid 程序的限定

对 setuid 与 setgid 的程序，使用上面一些内部的环境变量可能会带来一些安全问题。如果不对 setuid 与 setgid 的程序的动态库加载相关功能进行限制，那么可能会出现非法的提权操作。

举个简单的例子，linux 中的 passwd 命令就是一个设定了 setuid 的程序，设定此权限的目的在于让所有人都能够修改自己的密码，而 passwd 命令的属主与属组都是 root，当不同的用户在执行时将获得限制的 root 权限。如果使用动态库链接器 hook 某些函数，就可能会打破这种限制，达到提权的目的。

基于这样的问题，动态库链接器对内部一些可能修改正常程序执行行为的功能进行了严格限定，**一些环境变量将不会对具有 setuid 与 setgid 的程序生效。**