**网安实训第一周实验报告**

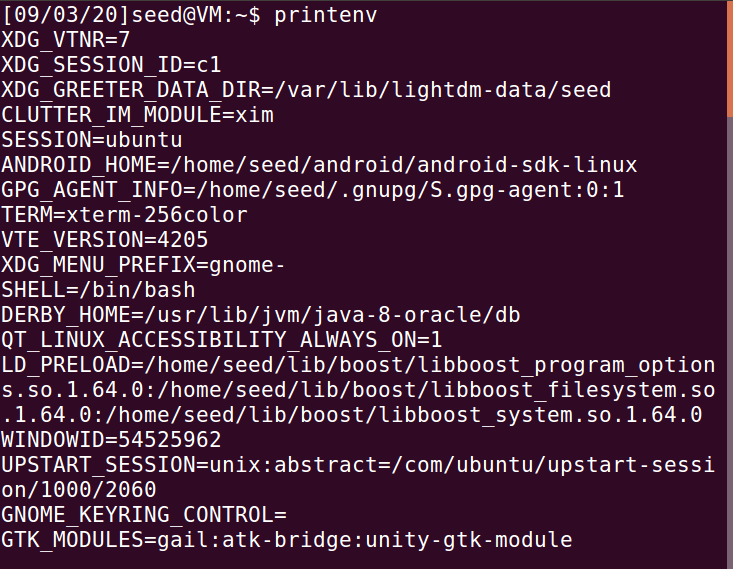
**——57118125施祺**

**Task 1**

任务要求：学会设置，删除环境变量，需要用到set，unset指令。

如下图所示，我们在linux中执行以下指令：

printenv:用于显示环境变量的值，输入并按回车，显示的是全部的环境变量。

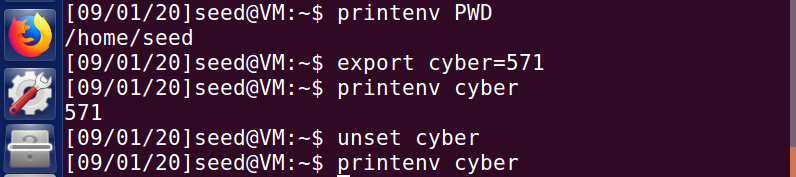


（图1-1）

我们也可以使用printenv去打印特定的环境变量。例如图1-2中，我们输入已有的环境变量”PWD“，即可显示其对应的值

下面我们利用export指令设置一个新的环境变量，姑且将称为cyber，并将其赋值为571，再将其打印，即可显示数值，表明新的环境变量设置成功。

接着，我们尝试用unset 指令删除cyber这个环境变量，再次打印，发现cyber无法显示其数值，表明环境变量已经删除。



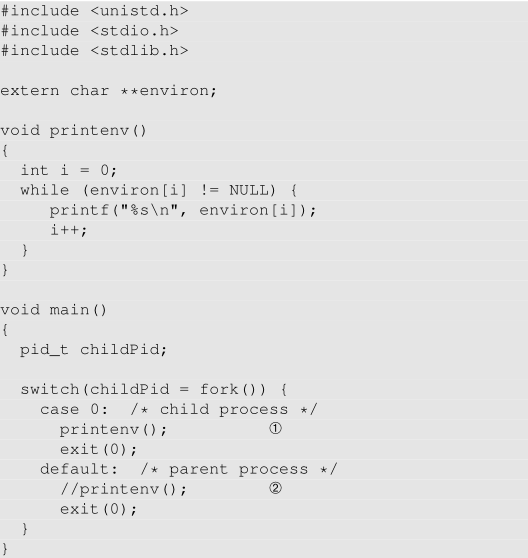
（图1-2）

**Task 2**

实验任务：将父进程的参数传递到子进程中，研究子进程如何从父进程中获得它的环境变量。

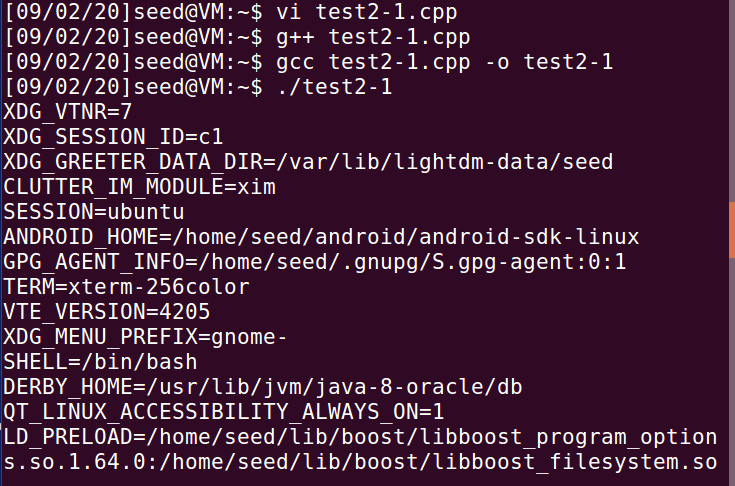
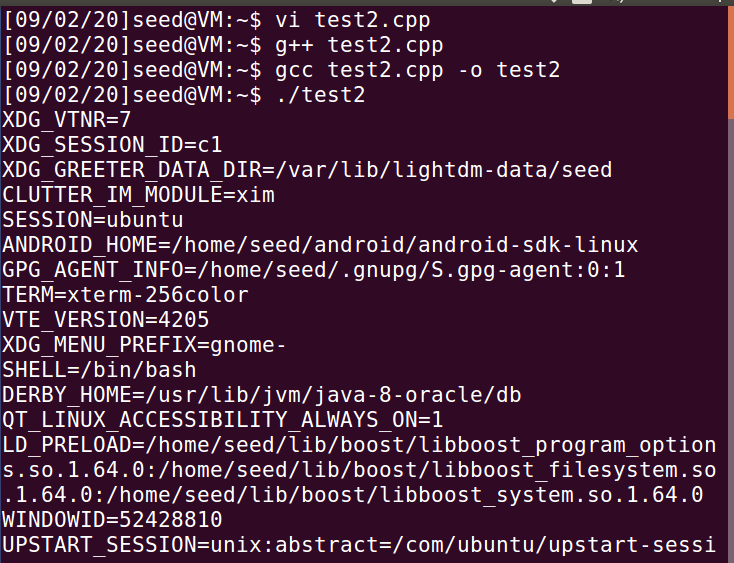
Step 1：

在linux中执行以下代码，并运行。



（图2-1）

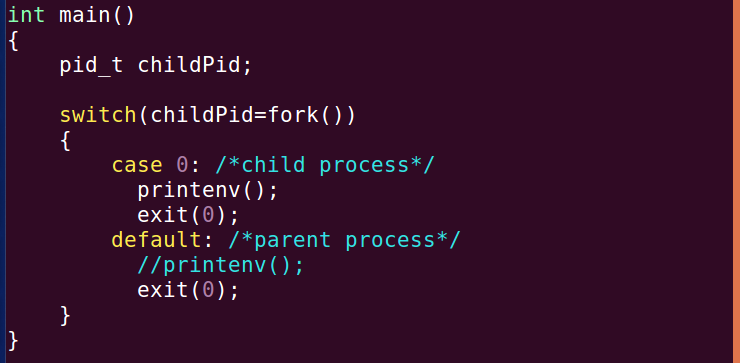
运行结果如图2-2所示，实现的功能为打印所有子进程的所有环境变量：



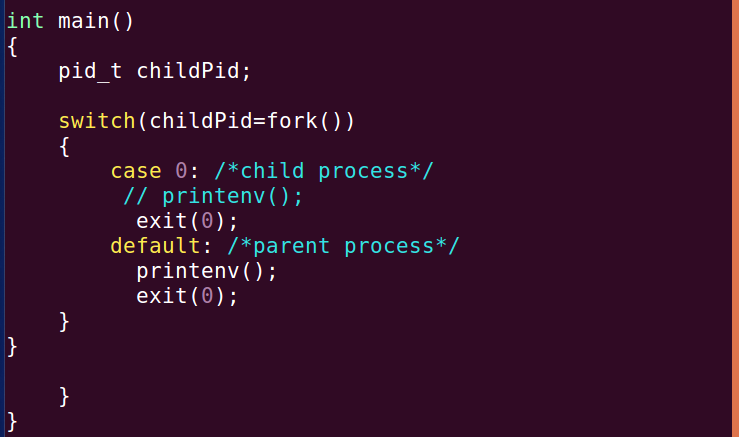
（图2-2）

**Step 2:**

注释掉case0中的printenv()，在default中加入printenv()，如图2-3，2-4所示



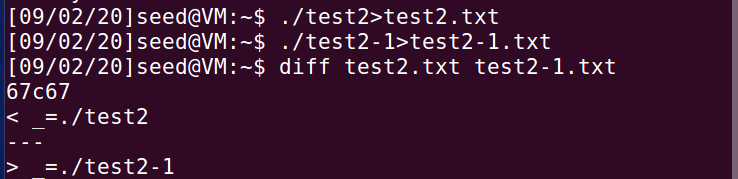
（图2-3：test2.cpp）



(图2-4：test2-1.cpp)

**Step 3:**

将两个程序的结果导出，并利用diff函数进行比较，从图2-5中我们可以看到，test2与test2-1的输出结果没有不同，说明子进程输出的环境变量与父进程一样，可以得出结论：子进程在产生时复制了父进程中的所有环境变量。



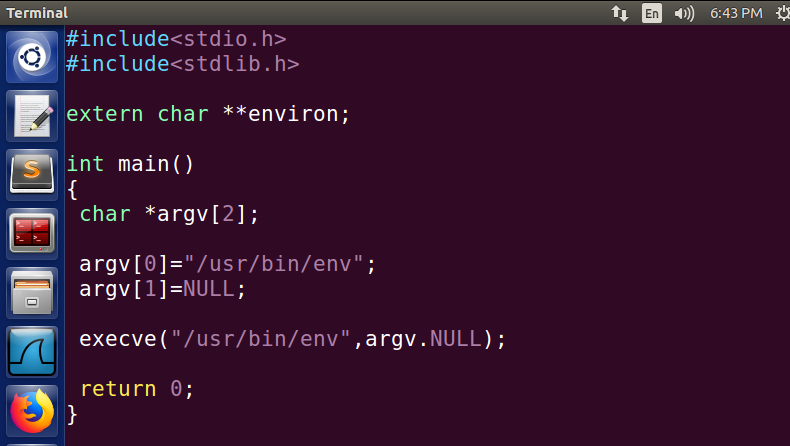
（图2-5）

**Task 3**

实验任务：当一个程序通过execve()运行时，将对环境变量产生怎样的影响。

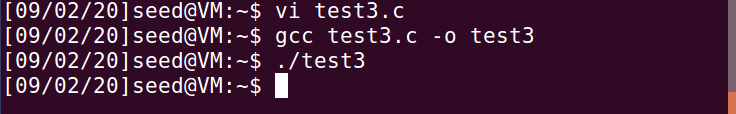
**Step 1:**

编译并运行图3-1所示的程序



（图3-1）

编译结果如图3-2所示，可以看到程序没有输出结果。



（图3-2）

**Step 2：**

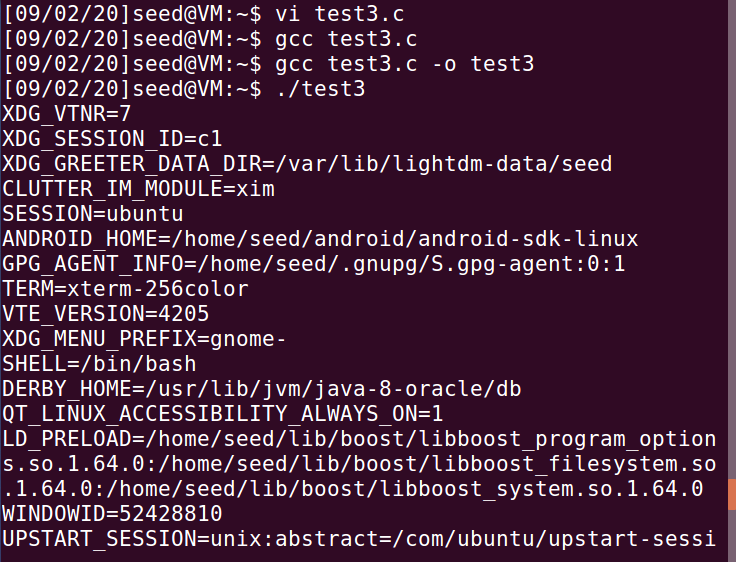
将图3-3中的代码更改为如图3-4所示，并进行编译，所得结果如图3-5所示。



（图3-3）



（图3-4）



（图3-5）

我们可以看到图3-5的输出结果，分析可知其功能为打印所有环境变量。

**Step 3**

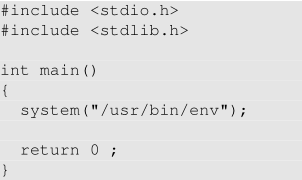
代码结果产生差异，原因来自于execve ()函数中参数environ对NULL的代换。全局环境变量environ的作用为打印所有环境变量。

execve(const char \*filename, char \*const argv[ ], char \*const envp[ ]); execve()用来执行参数filename字符串所代表的文件路径，第二个参数是利用数组指针来传递给执行文件，并且需要以空指针(NULL)结束，最后一个参数则为传递给执行文件的新环境变量数组。

程序在execve()中执行，获得它的环境变量的过程为：argv数组传递执行参数一所代表的执行文件“/usr/bin/env”，environ将自身指向的环境变量组再进行传递，从而程序获得它的环境变量。

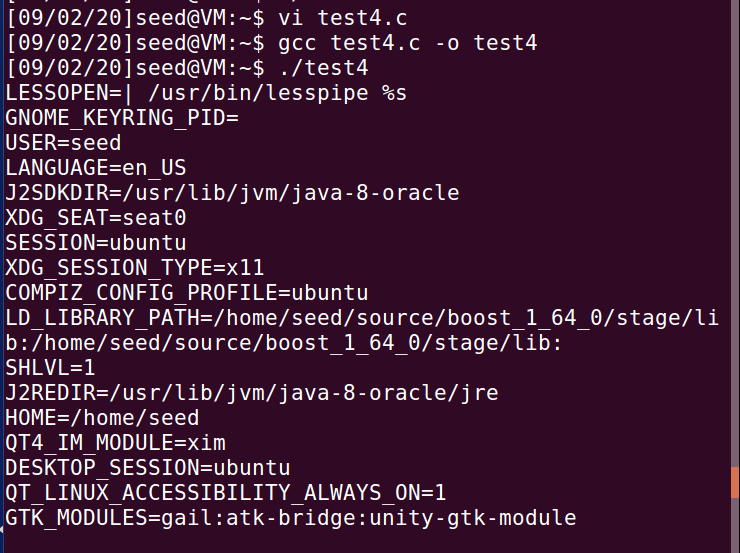
**Task 4**

实验任务：当一个程序通过system ()运行时，将对环境变量产生怎样的影响。



（图4-1）

我们编译如图4-1所示的代码，编译结果如图4-2所示



（图4-2）

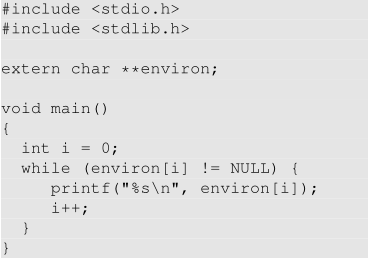
与Task 4进行对比，我们可以得出execve()与system()之间的差异。两者都为执行命令的函数，但execve()是直接执行这个命令，而system()则是通过shell去执行当前指令。因此，我们在使用system()函数时，相关进程的环境变量被传递到了新的程序中去执行，因此与execve相比，system少了传递的参数。

**Task 5**

实验任务：理解环境变量影响set-uid程序的过程

**Step 1**

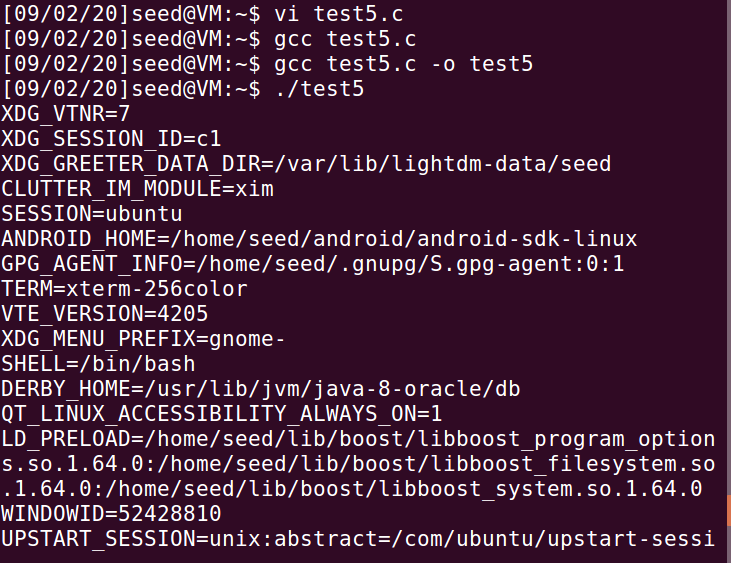
编写如图5-1所示的代码，用以指出当前进程中所有环境变量

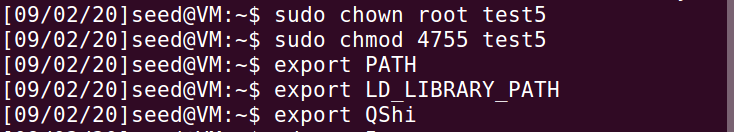


（图5-1）

**Step 2**

编译该程序，将其所有者更改为root，并让它成为一个Set-UID程序，操作过程如图5-2所示。





（图5-2）

**Step 3**

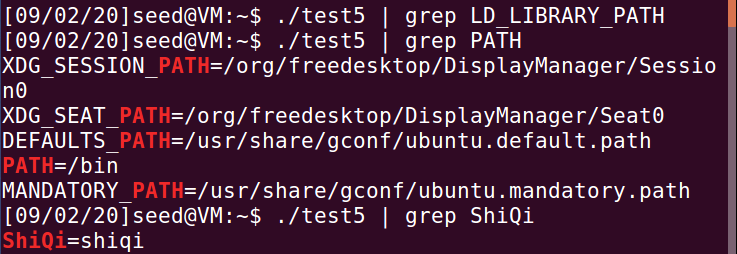
在一个非root权限的shell中，利用export命令去设立以下环境变量，如图5-3所示。





（图5-3）

运行程序，并在其中寻找我们设置的环境变量，结果如图5-4所示。



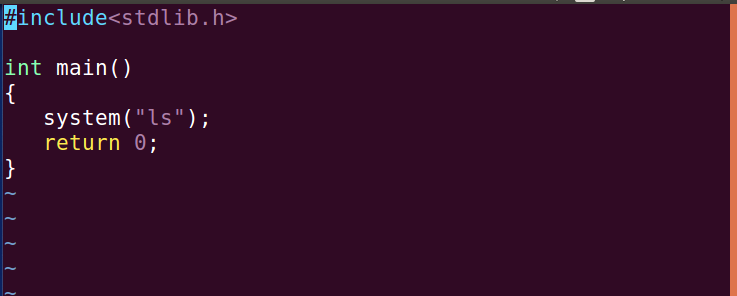
（图5-4）

我们可以发现在shell中可以找到我们新设置的环境变量以及赋值。由此我们可以发现Set-UID程序赋予用户特权的同时存在风险性，虽然程序按照自身逻辑运行，但用户仍可以利用环境变量的输入来影响程序的行为。

**Task 6**

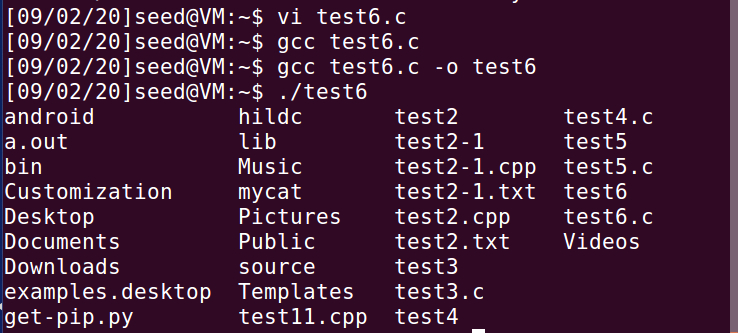
实验任务：利用system运行Set-UID程序的漏洞，通过PATH环境变量进行劫持。

首先我们在linux中输入如图6-1所示代码，代码的功能为通过system函数访问”ls”文件。



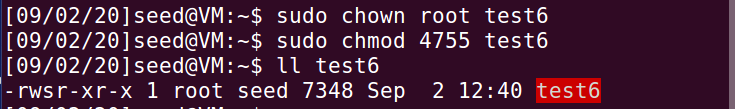
（图6-1）

运行结果如图6-2所示，功能为显示所有文件列表。



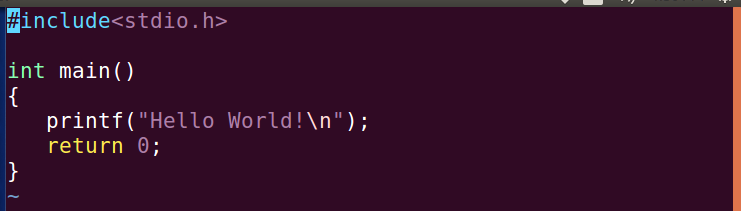
（图6-2）

下面我们更改程序所有者为root，并将其设置为Set-UID程序，如图6-3所示，通过ll命令，我们可以看到修改成功。



（图6-3）

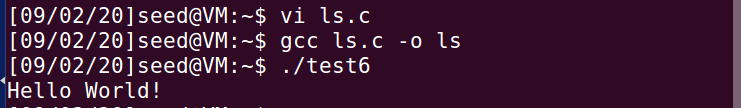
重新编写一个新的程序将其命名为“ls”，如图6-4所示。



（图6-4）

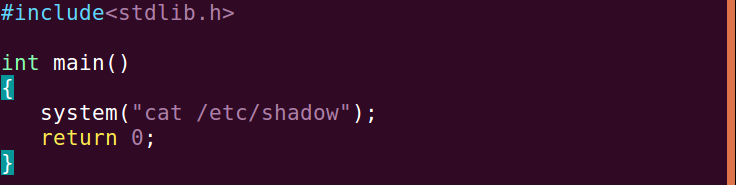
再次运行test6，结果发生变化，没有执行原先的程序而是运行了新的内容，说明system的文件内容被劫持。

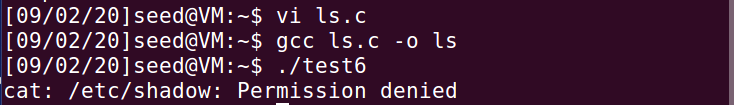




（图6-5）

在新版的ubuntu中，当访问的文件权限等级过高时，将自动显示无法访问。

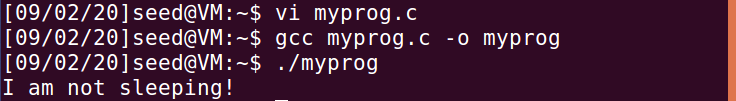
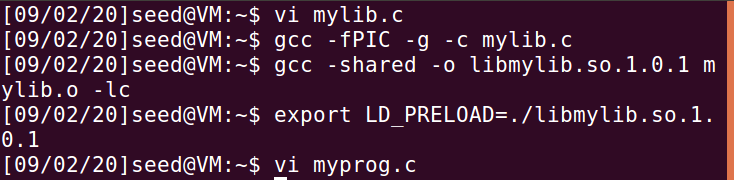
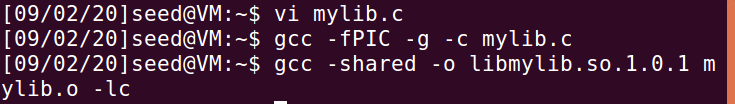




**Task 7**

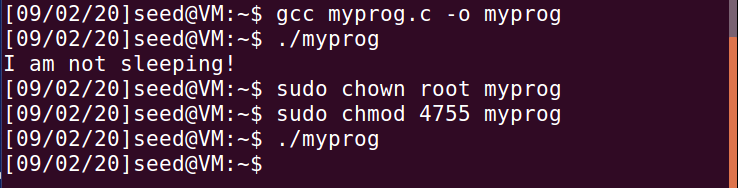
**Step 1**

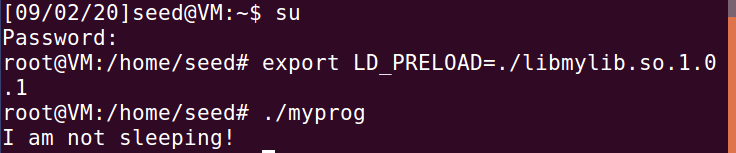
复制手册代码进行编译，设置代码mylib，并设置环境变量LD\_PRELOAD，再编译程序myprog。

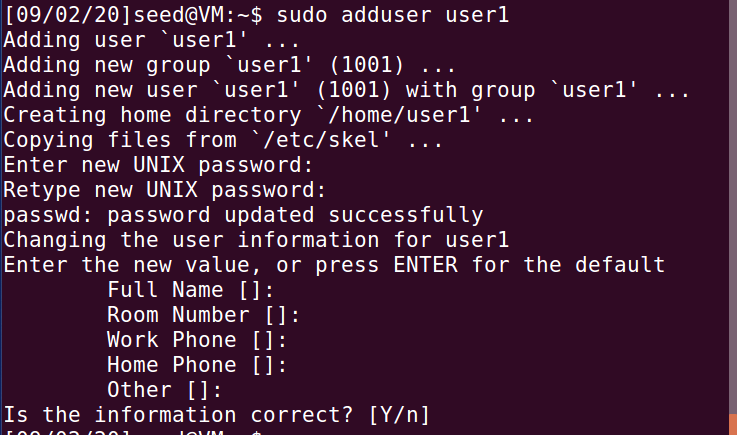


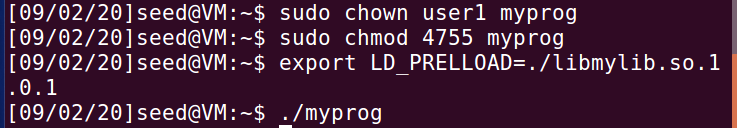
**Step 2**

在不同的情况下运行myprog，输出结果如下









**Step 3**

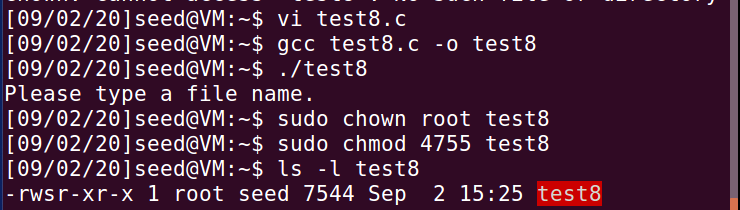
观察输出结果，我们可以发现四种情况中，在第一种和第三种两种情况下输出“I am not sleeping”，表明此时系统被劫持。我们分析以下情况：myprog作为普通程序，并且以普通用户权限运行；myprog作为set-UIDroot程序，在root账户中设立LD\_PRELOAD环境变量并且运行，这两种情况下会被劫持。由此我们可以知道只有在运行自己拥有的程序时才会使用LD\_PRELOAD环境变量和sleep函数，否则会忽略，所以导致了不同的实验结果。

**Task 8**

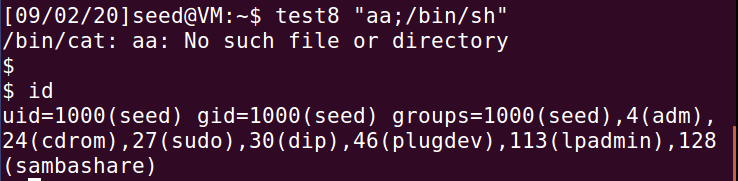
按照手册要求，编译上述程序

System()执行命令是危险的，用户可以提供参数给该命令，如果命令没有很好解析，那么用户输入的数据就会被误认为程序执行，从而发生危险。

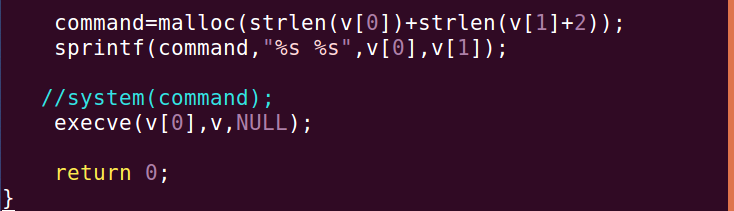
如果我是Bob，我可以通过输入特定的文件名，将输入的数据变成程序，从而达到修改程序的目的。

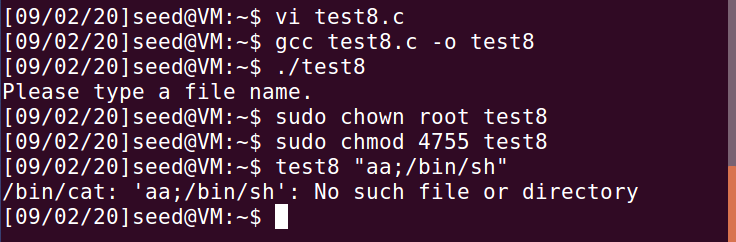


输入文件名“aa;/bin/sh” 分号是连接符，表示上一个shell结束，后面的bin表示我们手动开启一个shell，这表明我们可以进入shell执行命令，从而对文件进行写操作



将system()操作注释





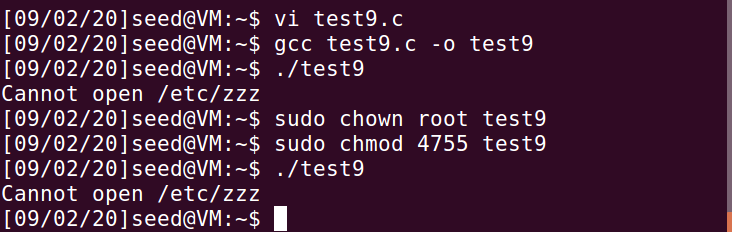
可以发现，使用execve()之后，”aa;/bin/sh”被整个当作一个文件名。上一步的攻击被成功避免。因为代码和数据被明确分开，用户输入数据无论如何也无法变成代码





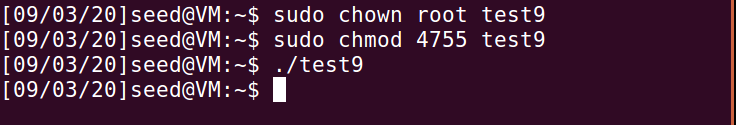
**Task 9**

复制手册代码并进行编译，结果如图9-1所示



（图9-1）





（图9-2）

如图9-2所示，我们打开/etc的目录，在其中添加了/zzz文件，再次运行，发现zzz文件成功创立，以此表明文件可以被修改。

分析原因，我们可以看到代码中句柄fd在完成后没有被删除，因此仍具有root权限，进而被用户利用完成对文件的修改。

