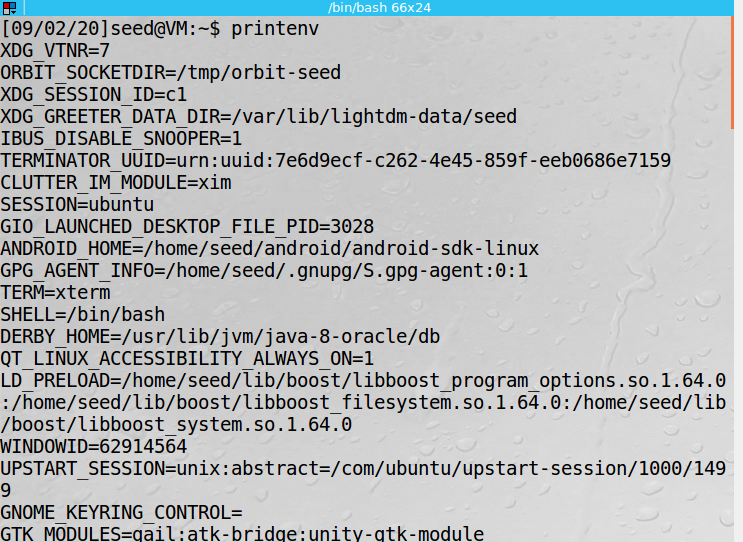
# lab1-report

**57117201王培丽**

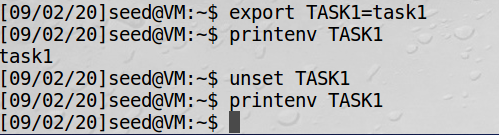
* **Task1: Manipulating Environment Variables**

1. 输出环境变量：



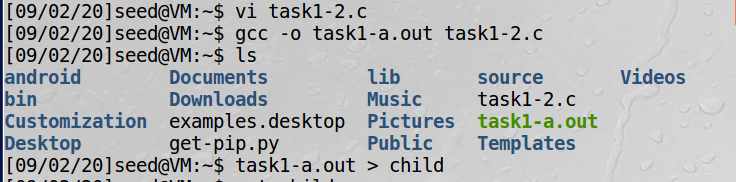


1. 设置环境变量：

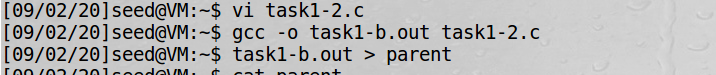


* **Task2: Passing Environment Variables from Parent Process to Child Process**

1. 编译task1-2.c，执行文件保存为task1-a.out，将输出保存到child文件



1. 修改task1-2.c内容，再次编译为task1-b.out，输出到parent文件



1. 比较结果



发现child和parent文件除了最后一行（75行），即文件名不一样，其他内容都一样，说明子进程继承了父进程的环境变量。

* **Task3: Environment Variables and execve()**

1. 编译程序task1-3.c，函数中环境变量参数为NULL，结果为空





1. 修改环境变量参数为environ，再次编译运行，结果如下（未全部截取）





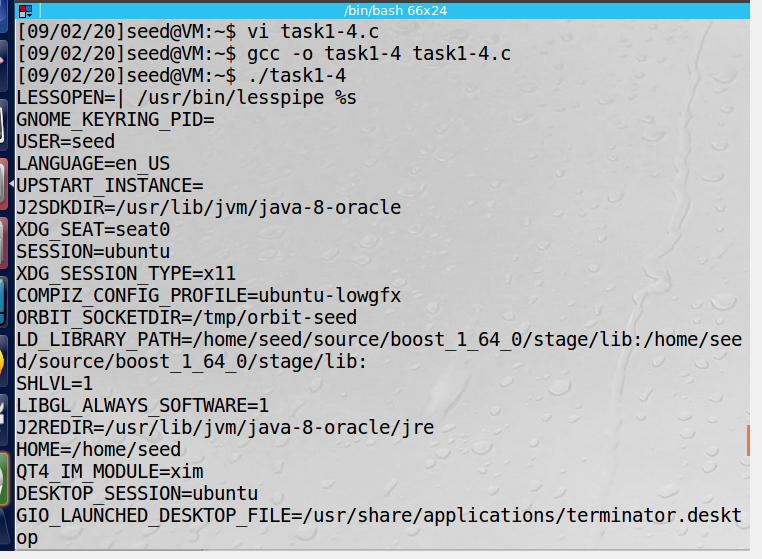
1. 结论：

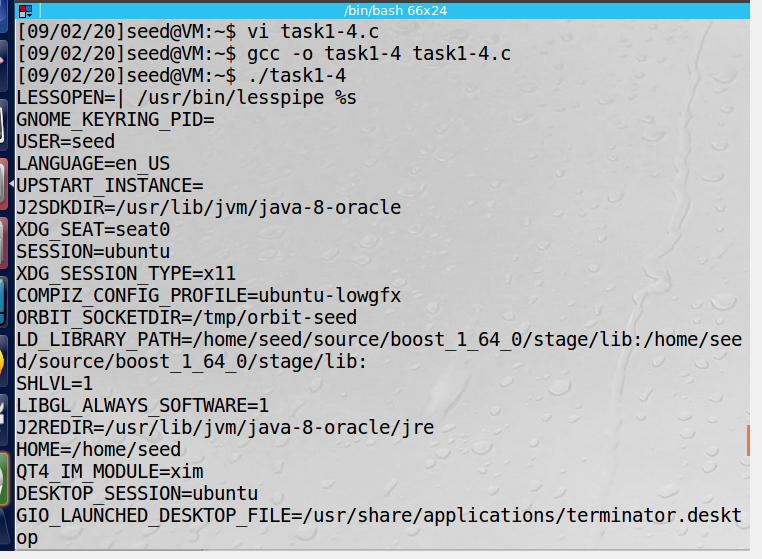
int execve(const char \* filename,char \* const argv[ ],char \* const envp[ ]);

execve()函数启动一个新程序，替换原有的进程，原进程的数据、堆栈等都会被这个新程序覆盖，所以其环境变量取决于使用execve()时通过最后一个参数传给它的新的环境变量，它并不会继承原进程的环境变量。

* **Task4: Environment Variables and system()**

编译task1-4.c，发现system()会传递环境变量（未全部截取）：





和execve()直接执行新程序不同，system()是通过调用/bin/sh来执行命令，并且将环境变量数组传递给/bin/sh，system()的执行过程实际上分为三步：

1 fork一个子进程

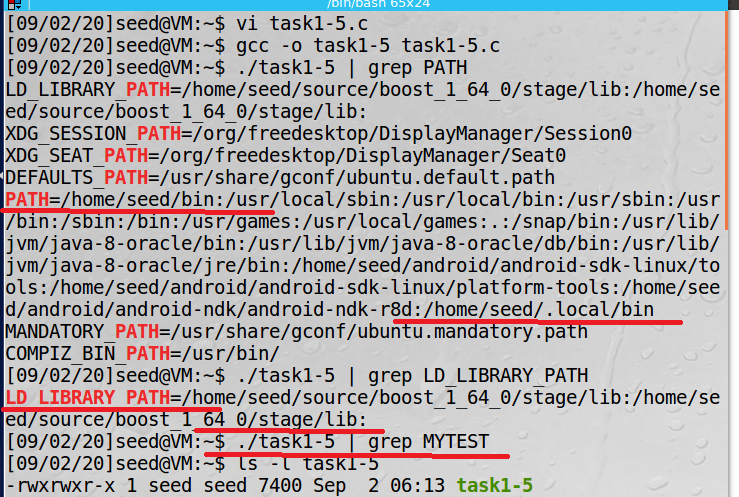
2 子进程调用exec执行command

3 父进程中调用wait等待子进程结束

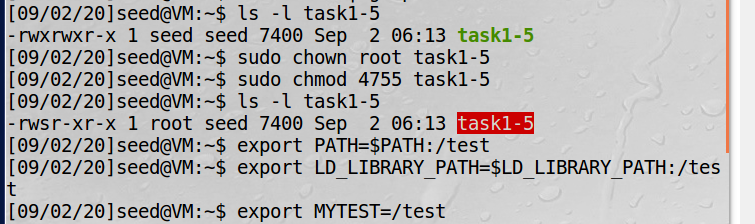
* **Task5: Environment Variable and Set-UID Programs**

1. 编译task1-5.c，生成task1-5，结果：

只查看下一步要设定的环境变量PATH、LD\_LIBRARY\_PATH、MYTEST（下一步要自定义的环境变量）：

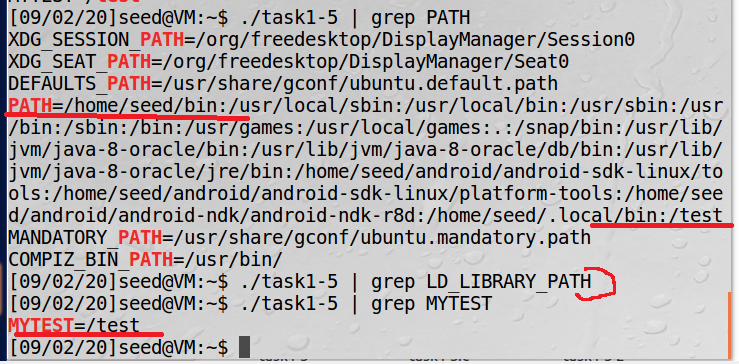


1. 设置task1-5为Set-UID root，并设置三个环境变量：



1. 再次运行task1-5，结果：

发现PATH和MYTEST的设置成功，出现了/test，但LD\_LIBRARY\_PATH环境变量没有了

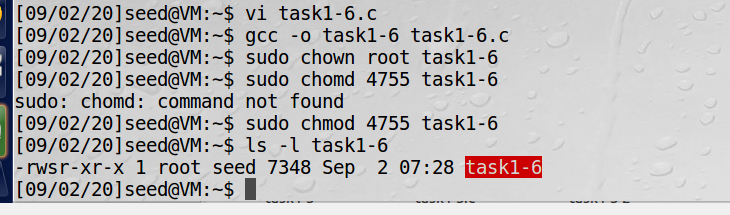


1. 结论：

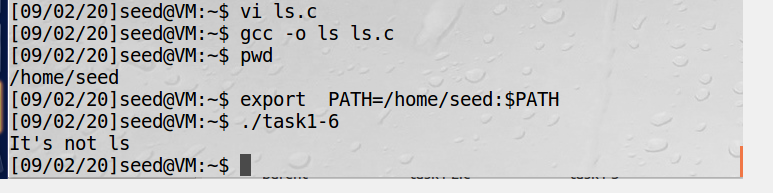
由上可得，Set-UID程序会继承用户的环境变量，Set-UID进程（子进程）会继承来自用户Shell（父进程）的环境变量。但出于安全考虑，在execl()以及其他一些exec函数在执行一个SUID程序时，会清除LD\_LIBRARY\_PATH环境变量，所以上面的实验中，仅PATH和MYTEST环境变量被继承，LD\_LIBRARY\_PATH被清除了。

* **Task6: The PATH Environment Variable and Set-UID Programs**

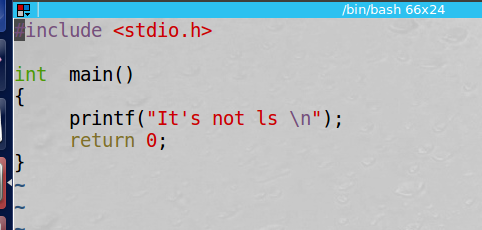
1. 编译代码生成task1-6，设置其所有者为root，设置Set-UID权限：



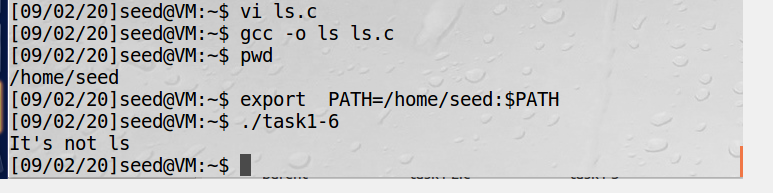
1. 在当前目录下编写一个自己的“ls.c”文件，并运行生成一个”ls”，并将当前目录加在环境变量PATH前面：



ls.c内容：

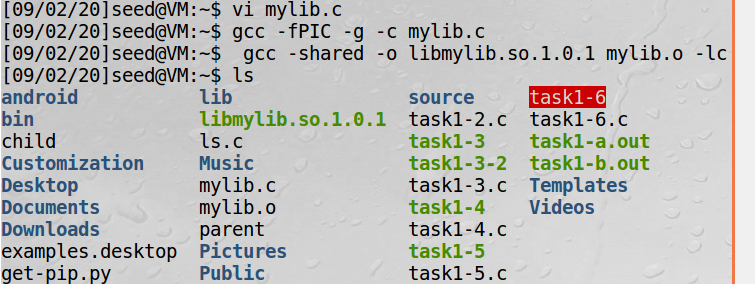


1. 再次运行程序task1-6，因为更改了PATH，会先寻找当前目录，就会先找到上面编写的“恶意的ls”，由此程序就会执行自定义的“ls”：



* **Task7: The LD PRELOAD Environment Variable and Set-UID Programs**

1. 编译代码mylib.c，生成一个动态链接库libmylib.so.1.0.1



1. 将LD\_PRELOAD环境变量设置为这个动态链接库libmylib.so.1.0.1



1. 在相同的目录下编译myprog.c程序

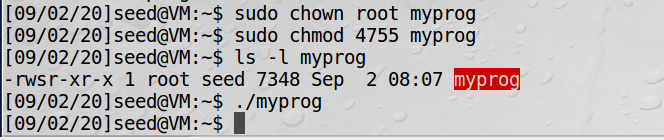


1. 在四种不同情况下运行myprog：

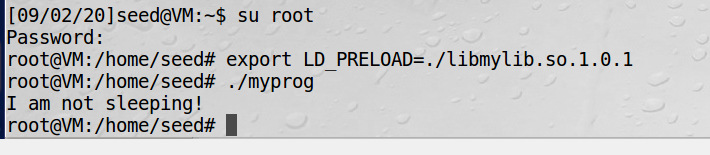
4.1 myprog为普通程序，以普通用户执行



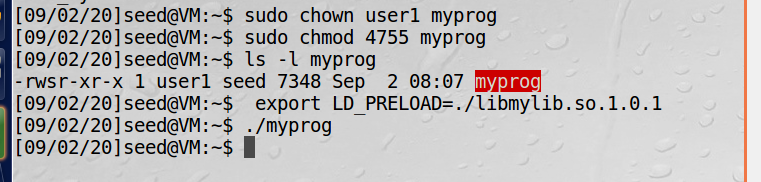
4.2 myprog为Set-UID root程序，以普通用户执行



4.3 myprog为Set-UID root程序，root身份下再设置一次LD\_PRELOAD环境变量并运行



4.4 myprog为Set-UID user1程序，另一个普通身份(seed)下设置LD\_PRELOAD并运行

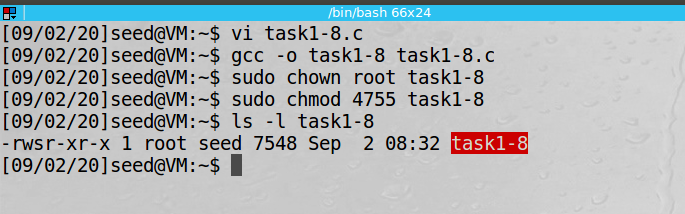


1. 结论

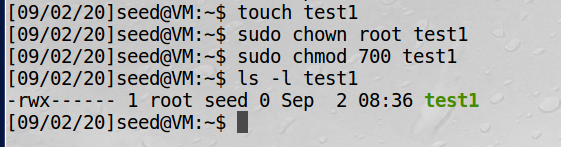
由上可知，当RUID=EUID时，设置的LD\_PRELOAD环境变量才会生效，否则就会忽略LD\_PRELOAD环境变量。所以seed身份运行owner是seed的myprog，root身份运行owner是root的myprog，环境变量的设置才有效，会运行重载的sleep函数，其他两种情况则不会。

* **Task8: Invoking External Programs Using system() versus execve()**

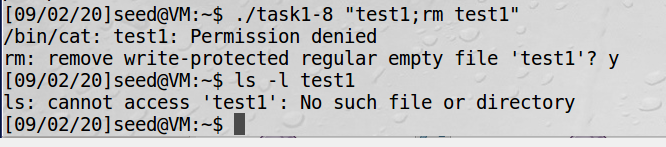
1. 编译程序task1-8.c生成task1-8，将其设置为Set-UID root



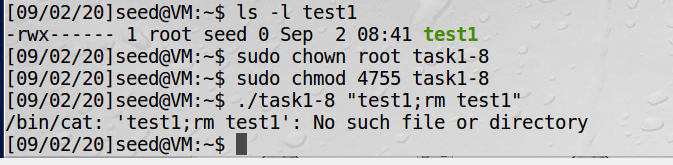
1. 新建一个测试文件test1，将其权限设为非root用户均不可读写执行



1. 作为Bob，在执行task1-8时，输入恶意的语句作为文件名“test1; rm test1”，发现命令生效，test1已经被删除



1. 将task1-8.c中的system(command)注释掉，改用execve()，再次编译运行，设置为Set-UID root.。重复3中步骤，恶意输入“test1; rm test1”已经不能生效。



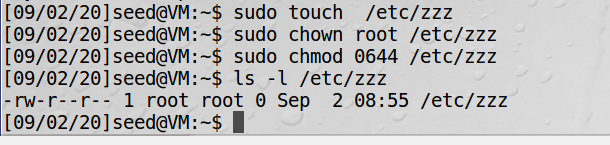
1. 结论

使用system()时，system(command)，调用了/bin/sh -c command来执行command所定义的命令，输入“test1;rm test1”，会将其当作两个命令来执行：原程序定义的 /bin/cat test1以及 rm test1 。即使seed本没有权限删除test1，在task1-8的执行过程中也可以将其删除。

而在使用execve()时，execve(v[0], v, NULL)中将用户的输入“test1; rm test1”当作一个整体，作为文件名传给参数“v”，只会执行v[0]定义的/bin/cat来查看v定义的文件，所以会报错找不到“test1; rm test1”这个文件（目录）。

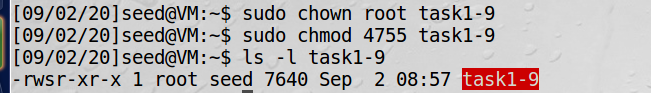
* **Task9: Capability Leaking**

1. 生成一个所属者为root，权限为0644的文件/etc/zzz



1. 编译程序task1-9.c生成task1-9，将其设置为Set-UID root





1. 以普通用户身份执行task1-9，发现/etc/zzz文件被改写了



1. 结论

因为在程序中，先打开了文件，随后虽然取消了特权，但文件还未关闭，造成了权限泄漏，导致seed用户仍能以root的身份对/etc/zzz文件做写入操作。