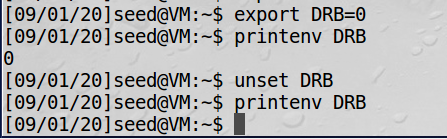
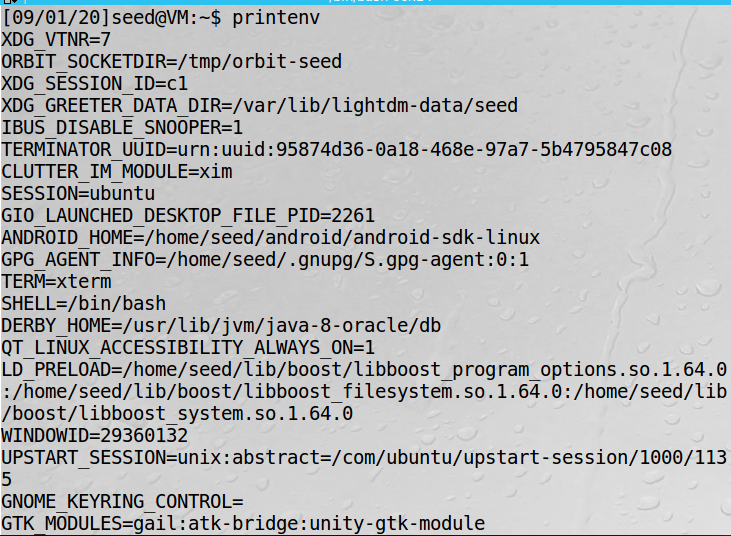
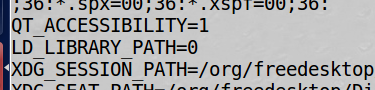
Environment Variable and Set-UID Program Lab

57118138 李嘉怡

**Task 1: Manipulating Environment Variables**

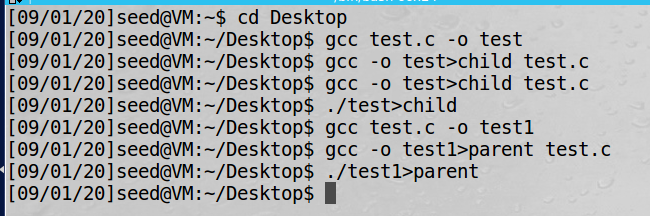
用printenv能显示所有环境变量，也可以在后面加变量名来显示某个环境变量的值，还可以用export和unset语句自己定义和删除环境变量。

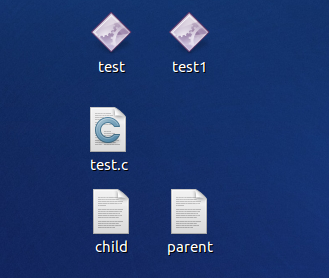




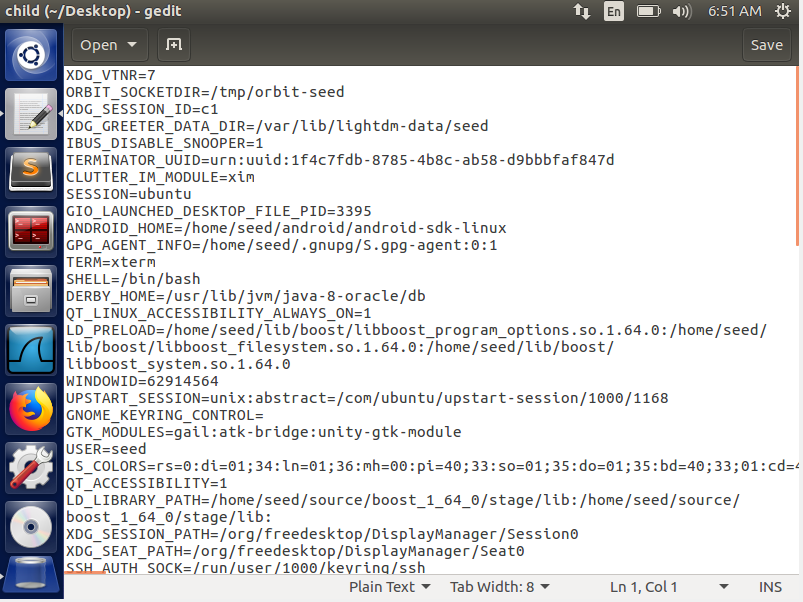
**Task 2: Passing Environment Variables from Parent Process to Child Process**

分别注释掉子进程和父进程中的printenv();语句，分别生成可执行文件test和test1，并将执行结果生成文件。



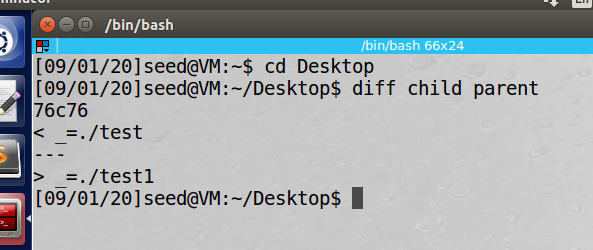


child：



parent：

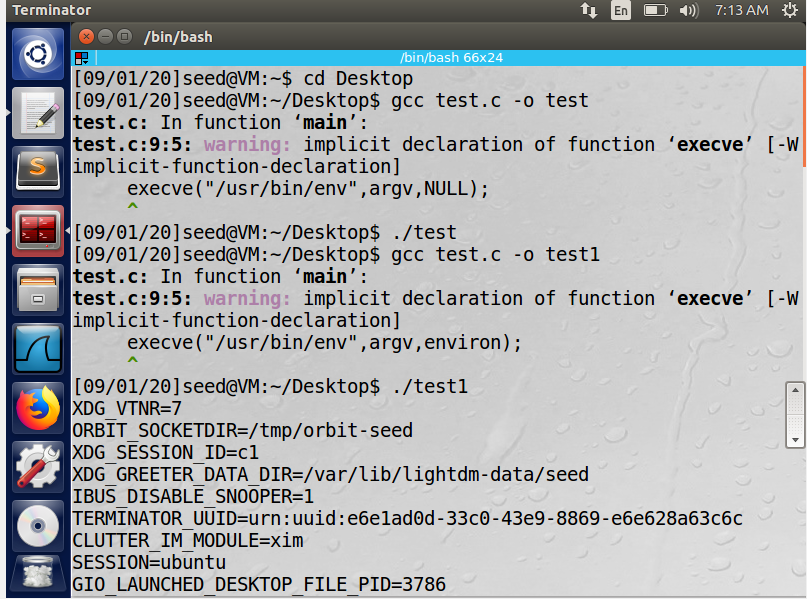




通过diff语句可知，child和parent的内容中除了对应的可执行文件名字不同，其他都相同。父进程通过fork（）生成子进程时，是完全将父进程拷贝了一份，然后两个进程并发执行，它们除了进程的pid不同以外其他都相同，因此子进程完全继承了父进程的环境变量，二者环境变量完全相同。

**Task 3: Environment Variables and execve()**

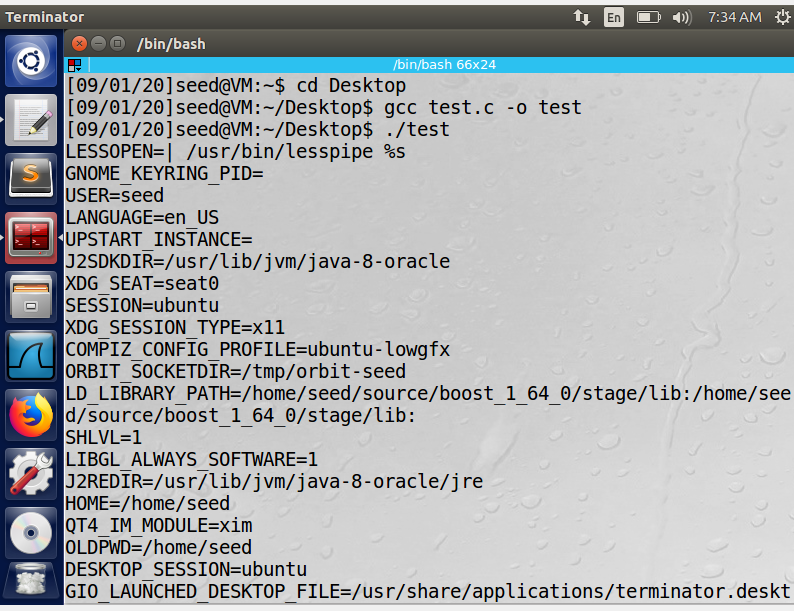
分别令execve()函数中第三个参数为NULL和environ，进行编译。



发现第一次编译（参数为NULL）没有输出，而第二次编译（参数为environ）输出了所有环境变量。execve()函数最后一个参数为传递给执行文件的新环境变量数组，因此，与fork()不同，用execve()创建新的进程的时候环境变量是通过参数传递而不是自动继承的，如果传递的参数为NULL则不传递环境变量。

**Task 4: Environment Variables and system()**

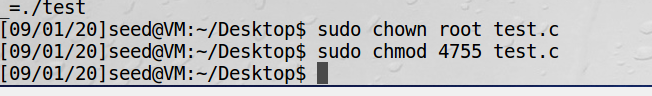
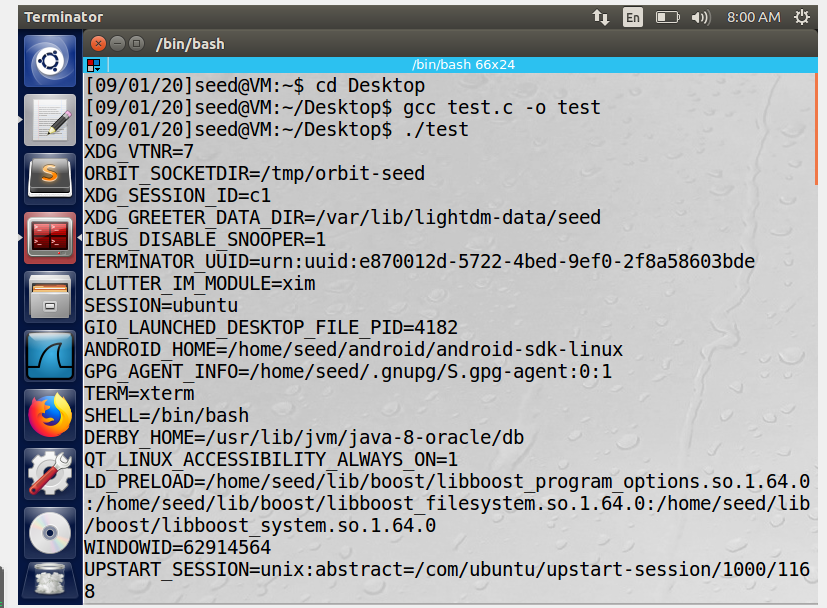
输入程序并编译运行，输出得到所有的环境变量。



因此证明了system()使用execl()执行/bin/sh;execl()调用execve()，并将环境变量数组传递给它，使用system()，调用进程的环境变量被传递给新程序/bin/sh。System()函数虽然没有在参数中传递环境变量，但是依然达到了传递了环境变量的效果。

**Task 5: Environment Variable and Set-UID Programs**

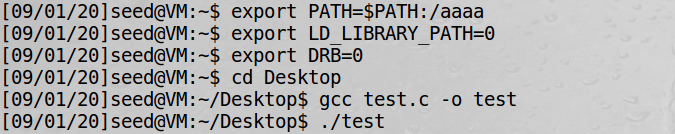
执行程序，输出了当前进程中所有的环境变量。将其所有权改为root，并使其成为Set-UID程序。



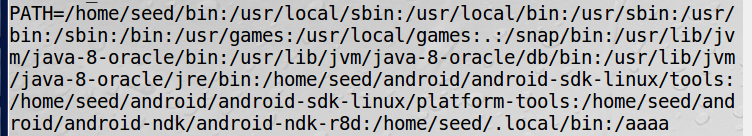
修改为set-uid程序以后，发现图标的右下角多了一个锁。

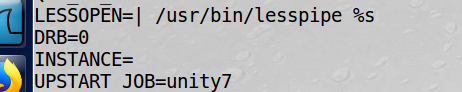
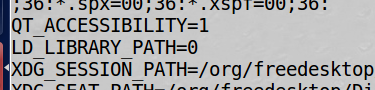


在shell中用export设置三个环境变量。



再次编译执行test.c，发现能够在列出的环境变量中找到刚才自己定义的环境变量PATH，LD\_LIBRARY\_PATH和DRB。

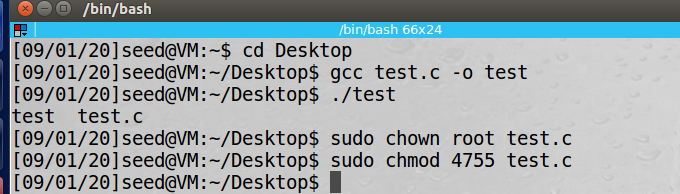


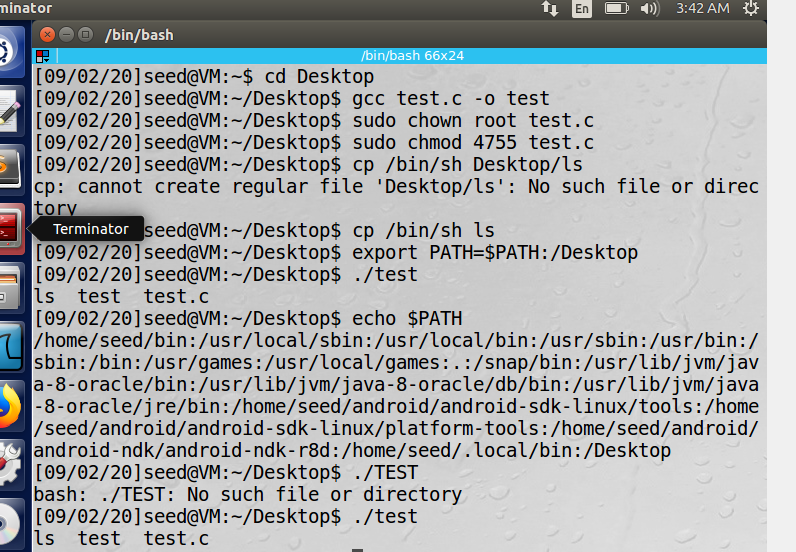
因此可以证明，在父进程shell中设置的所有自己定义的环境变量都进入了set - uid子进程。

**Task 6: The PATH Environment Variable and Set-UID Programs**

编译并运行该程序，并使它成为set-uid程序。

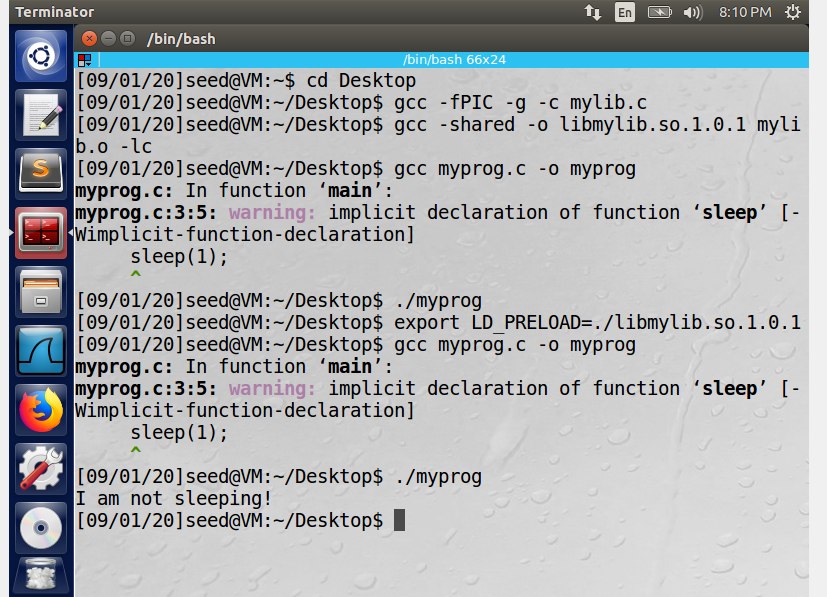


先使/bin/目录下的sh 符号链接到zsh，而不是bash，然后把/bin/sh拷贝到/tmp目录下面重命名为ls，将环境变量PATH设置为当前目录/tmp，运行编译的程序test。就可以获得root权限。

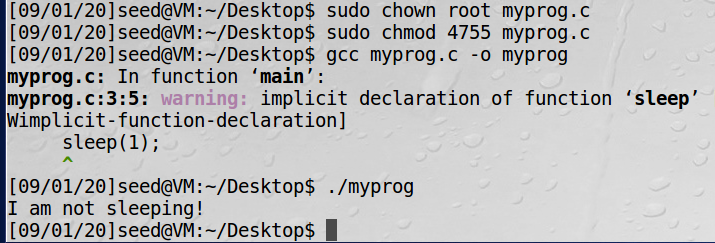


**Task 7: The LD PRELOAD Environment Variable and Set-UID Programs**

##### ①把myprog.c编译成一个普通用户下的程序在普通用户下运行。可见如果没有增加环境变量，执行的是系统原本的sleep()函数，定义环境变量之后执行的是自己定义的的sleep()函数。

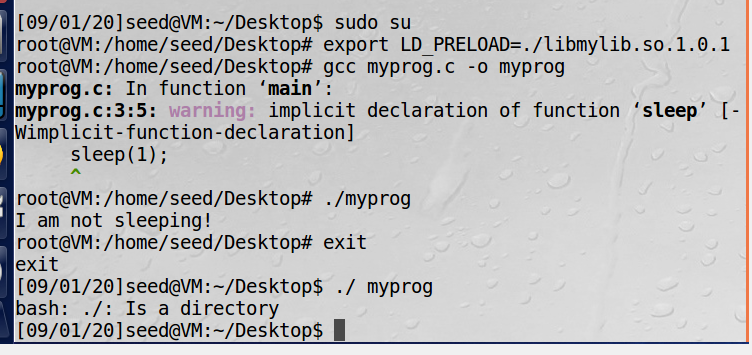


##### ②在一个普通用户下把myprog.c编译成一个Set-UID 普通用户的程序在普通用户下运行。在这种情况下，执行的是自己定义的的sleep()函数。



③将myprog设置为一个Set-UID root的程序，在root中再次导出LD\_PRELOAD 环境变量并运行它。此时执行的是自己定义的的sleep()函数。

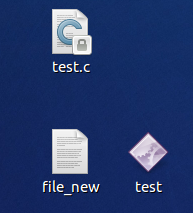
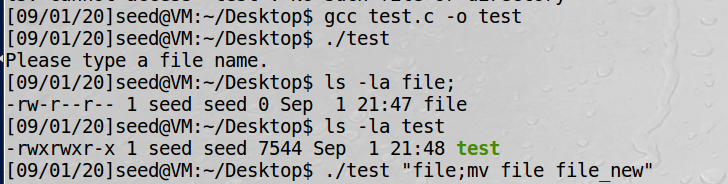
④退出root，myprog是一个Set-UID root的程序，在普通用户中运行它，在这种情况下，会忽略LD\_PRELOAD环境变量，不重载sleep()函数，使用系统自带的sleep()函数。



由以上四种情况可见：只有用户自己创建的程序自己去运行，才会使用LD\_PRELOAD环境变量，重载sleep()函数，否则的话忽略LD\_PRELOAD环境变量，不会重载sleep()函数。

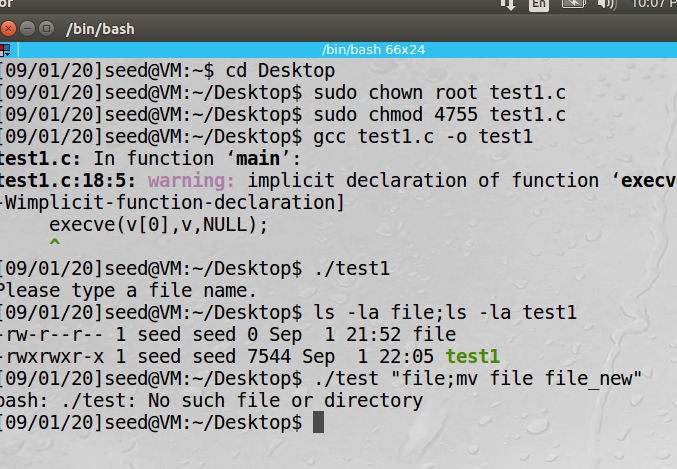
**Task 8: Invoking External Programs Using system() versus execve()**

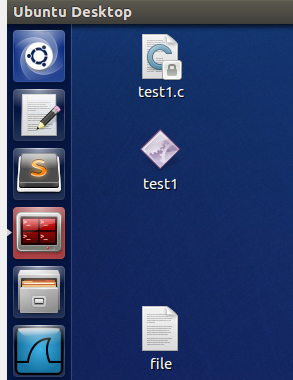
①编译该程序，使它成为root的Set-UID程序，程序使用system()调用命令。这个命令不安全，Bob可能会出于好奇或者个人利益驱使阅读或者修改只有root用户才可以运行的一些文件。



在这里，file文件本来只有root用户有读写权限，但普通用户通过运行该程序，阅读并重命名了file文件，使其变成file\_new。

②注释掉对system()调用的语句，使其执行execve()函数。

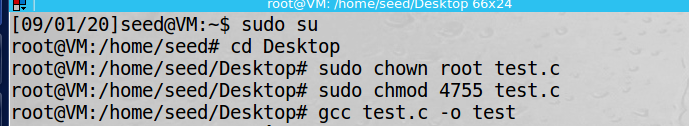


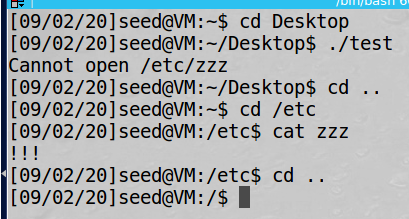
此时不能再修改file的名字了，证明前一步中的攻击在这里是无效的。

调用system()时，这种攻击有效，是因为system()函数调用/bin/sh，链接至zsh，具有root权限执行了cat file文件后，接着执行mv file file\_new命令。而当程序调用execve()时, execve()函数会把file; mv file file\_new 看成是一个文件名，系统会提示不存在这个文件。

**Task 9: Capability Leaking**

编译该程序，使其成为root下的Set-UID程序。以普通用户的身份运行程序。







发现zzz文件被修改了，原因在于设置uid前，zzz文件就已经被打开了。只要将语句setuid(getuid())移至调用open函数之前，就能避免这个问题。