**信息安全实验报告**

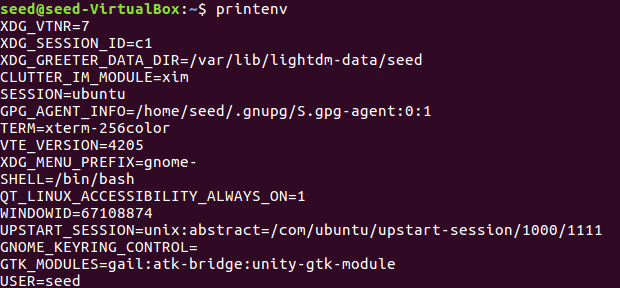
**Lab 2 Set-UID and EVA**

**孙铁**

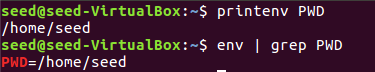
**SA20225414**

**Task 1**

输入 printenv 或 env 打印所有的环境变量 (部分截图如下)：



输入printenv 或 env | grep 后接环境变量名打印特定环境变量



输入 export 后接环境变量名修改特定环境变量



输入 unset 后接环境变量名取消设置



此时发现环境变量表中的PWD被删除：

**Task 2**

分析代码，可以看出其作用是通过fork函数创建子进程，并打印其环境变量。

将代码命名为task2.c并编译运行，将结果输出到child文件



注释掉 ➀ ，并将 ➁ 注释取消，再次编译运行，将结果输出到child2文件



比较child与child2



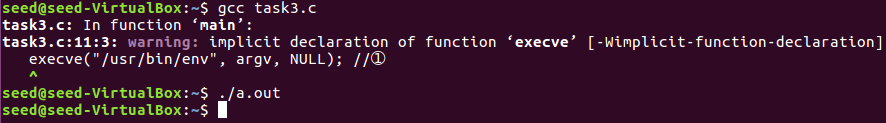
根据结果可以看出，child与child2没有不同

本部分实验其实是分别获取了父进程与其调用fork函数创建的子进程的环境变量，并进行对比。根据结果可以看出，子进程会继承父进程所有的环境变量。

**Task 3**

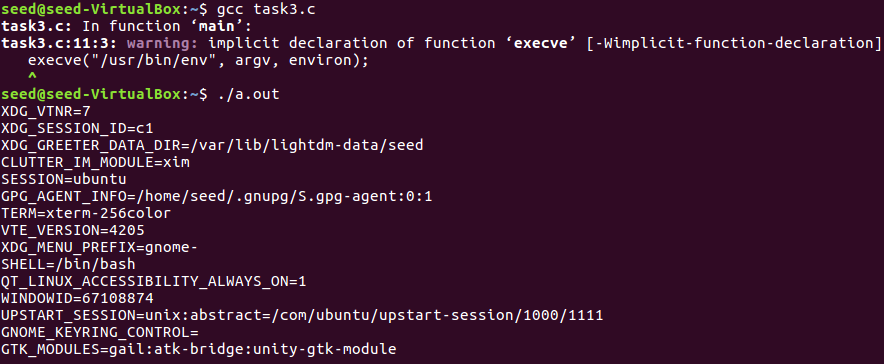
分析代码，其作用是调用execve函数开启一个新进程。

将代码命名为task3.c并编译运行



没有输出

将 ➀ 修改为“ execve("/usr/bin/env", argv, environ); ”，并编译运行



输出了环境变量 (部分截图)

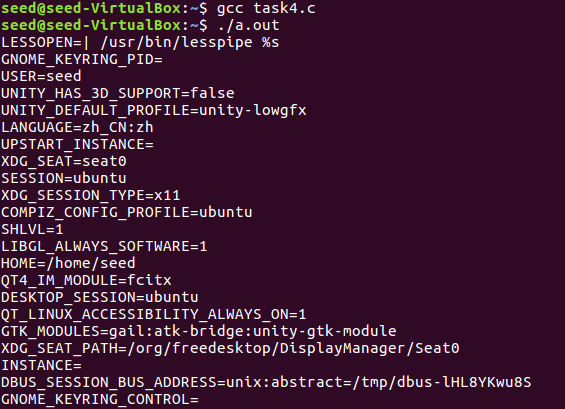
execve函数是系统调用函数，进程通过execve函数开启新进程时，原进程的内存会被新进程数据覆盖，这就导致原进程的环境变量将会丢失。

execve函数有三个参数：要运行的新程序路径；新程序的所有参数；新进程的环境变量。新进程可以根据execve函数的参数来获取环境变量：当execve函数第三个参数为NULL时将不传递环境变量；将execve函数的第三个参数由NULL改为environ，则可以将调用execve函数的进程的所有环境变量传递给新进程。

**Task 4**

分析代码，其作用是调用system函数开启一个程序位置在 /usr/bin/env 的进程。

将代码命名为task4.c并编译运行：



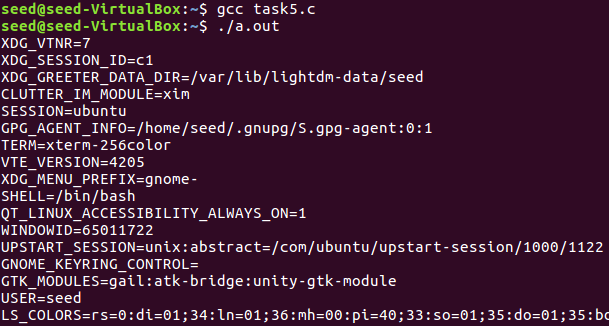
输出了新进程的环境变量。

system函数调用execl函数，然后execl函数调用execve函数传递环境变量。

**Task 5**

分析代码， 其作用是打印全部的环境变量。

将代码命名为task5.c并编译运行 (部分截图)：



将此程序拥有者改为root用户，并把他变为一个Set-UID程序





发现此时的程序图标右下角多了一个锁

设置环境变量PATH，LD\_LIBRARY\_PATH，ANYNAME：



运行Set-UID程序a.out，会生成一个子进程执行此程序

可以看到PATH与ANYNAME都被修改：



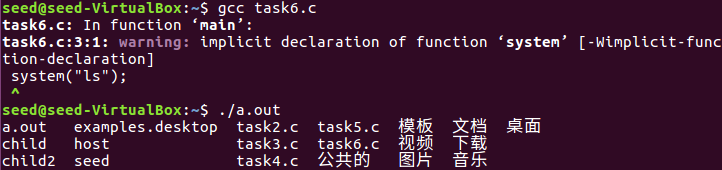


但并未出现LD\_LIBRARY\_PATH，说明子进程的LD\_LIBRARY\_PATH环境变量未被修改。

**Task 6**

分析代码，其作用是通过system函数调用“ls”指令，列出当前目录的所有文件。

将代码命名为task6.c并编译运行



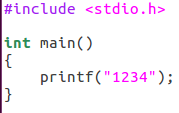
将此程序拥有者改为root用户，并把他变为一个Set-UID程序



考虑如何让a.out运行时并不执行“ls”命令，而是转为执行一个自定义的程序：

Shell程序运行命令时，如果没有提供命令的具体位置，则将在PATH环境变量中搜索命令。通过修改PATH环境变量，在PATH的头部进行修改，插入一个与命令名称相同的“ls”程序的路径。这样程序创建进程时就会先搜索修改所指向的目录，调用预先准备好的“ls”程序。

编写ls程序



并编译ls.c为可执行文件ls.out



由于 /bin/dash中对这种Set-UID程序的攻击有所防护。为了取得攻击效果，需要将 /bin/sh 链接到另一个没有这种防护措施的shell程序 bin/zsh。



在PATH环境变量头部加入代表当前目录的“.”

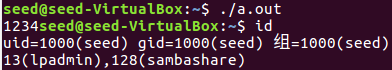


运行a.out

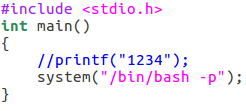


成功运行了“ls”程序

查看权限，没有显示有效用户：



更改ls.c文件为：



编译为可执行文件ls.out

运行a.out并查看权限：



可以看到有效权限为root，说明此shell确实以root权限运行。

**Task 7**

分析代码1， 其作用是重写了标准libc共享库中的sleep函数。

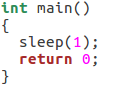
将其命名为mylib.c并使用如下命令编译



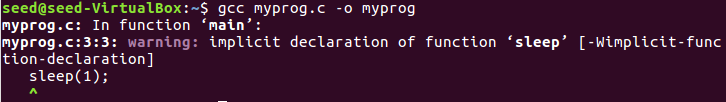
修改LD\_PRELOAD



编写myprog.c如下：



将其编译为可执行文件myprog



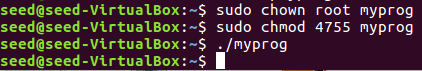
接下来尝试在不同情况下运行myprog:

1. 以普通用户身份运行myprog程序：



调用了重写函数

1. 将程序myprog拥有者设为root，并将其变为Set-UID程序，以普通用户身份运行：



睡眠1s，未调用重写函数。

1. 在root账户下设置LD\_PRELOAD，将程序myprog拥有者设为root，并将其变为Set-UID程序，以root身份运行：



调用了重写函数。

1. 设置LD\_PRELOAD，将程序myprog拥有者设为user1，并将其变为Set-UID程序，用另一个非root用户user2运行myprog：

创建新用户newuser：



将myprog程序拥有者设为newuser，并将其变为Set-UID程序：



以seed用户身份运行：



睡眠1s，未调用重写函数。

根据以上四种情况下的结果可以看出：

当Set-UID进程的真实运行用户和有效用户不同时，进程将会忽略环境变量LD\_PRELOAD，所以不会发生重写。

下面针对此结论进行实验验证：

env是打印环境变量的程序，将其制作一个拷贝env1。将env拥有者设为root，并将其变为Set-UID程序：



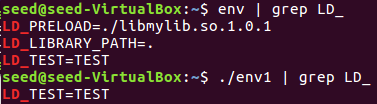
此时env1的有效用户为root：



修改环境变量LD\_PRELOAD，LD\_LIBRARY\_PATH以及一个自定义的参考变量LD\_TEST:



以seed用户分别运行env与env1：



可以看到LD\_PRELOAD，LD\_LIBRARY\_PATH被忽略。

**Task 8**

分析代码，其作用是调用cat命令根据输入文件的名称来查看指定文件，代码本身只有读功能，并没有修改功能。但由于代码使用system函数来调用cat程序，这就导致用户可以通过间接修改system函数的输入参数来获得root权限的shell。

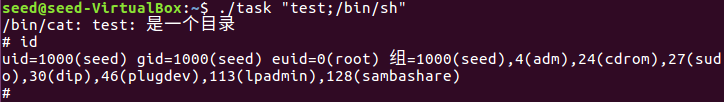
将代码命名为task8.c并编译为可执行文件task，将其转为拥有者为root的Set-UID程序：



为了得到实验效果，依然需要将 /bin/sh 链接到另一个没有防护措施的shell程序 bin/zsh：



运行task程序并输入“test;/bin/sh”，此时system函数就会执行“bin/cat/test”以及“bin/sh”两个命令：

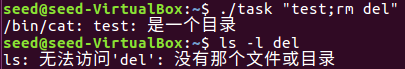


可以看到获得了一个有效权限为root的shell。

创建一个文件del，并将拥有者改为root：



运行task程序并输入“test;rm del”：



成功删除del

将system函数替换为execve函数





发现同样的攻击方式不能提权了。

system函数执行command的过程中需要调用fork函数生成子进程，然后将command输入子进程的shell来执行。由于shell支持一行中输入以分号分隔的两个命令，这就出现了一个被会攻击的漏洞。

而execve函数会直接请求操作系统执行输入的指定命令，哪怕输入了额外的指令，也只会被当作一个参数。

**Task 9**

分析代码，其作用是打开文件并通过文件描述符fd对文件进行操作，并在操作结束后，关闭特权，销毁描述符。

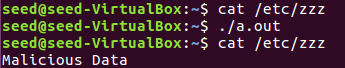
将代码命名为task9.c并编译，将a.out程序转化为拥有者为root的Set-UID程序：



创建文件 /etc/zzz：



运行a.out程序并查看 /etc/zzz中内容：



可以看到已经成功在文件中写入数据。

这是因为程序在打开文件完成操作之后，虽然调用setuid函数关闭了特权，但子进程中的文件操作符并没有被销毁，导致非特权进程依然可以对文件进行修改。