Lab1 Report

姓名：银皓然　学号：57119116　完成日期：2021.7.6

一、实验目的（Task1-6 Task8）

通过各类任务，学习并了解有关操作系统中环境变量的作用、如何操纵环境变量以及环境变量如何影响计算机上进程的执行。

2.1 Task 1: Manipulating Environment Variables

In this task, we study the commands that can be used to set and unset environment variables. We are using Bash in the seed account. The default shell that a user uses is set in the `/etc/passwd` file (the last field of each entry). You can change this to another shell program using the command `chsh` (please do not do it for this lab). Please do the following tasks:

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

In this task, we study how a child process gets its environment variables from its parent. In Unix, `fork()` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent; however, several things are not inherited by the child (please see the manual of `fork()` by typing the following command: `man fork`). In this task, we would like to know whether the parent's environment variables are inherited by the child process or not.

2.3 Task 3: Environment Variables and `execve()`

In this task, we study how environment variables are affected when a new program is executed via `execve()`. The function `execve()` calls a system call to load a new command and execute it; this function never returns. No new process is created; instead, the calling process's text, data, bss, and stack are overwritten by that of the program loaded. Essentially, `execve()` runs the new program inside the calling process. We are interested in what happens to the environment variables; are they automatically inherited by the new program?

2.4 Task 4: Environment Variables and `system()`

In this task, we study how environment variables are affected when a new program is executed via the `system()` function. This function is used to execute a command, but unlike `execve()`, which directly executes a command, `system()` actually executes `"/bin/sh -c command"`, i.e., it executes `/bin/sh`, and asks the shell to execute the command.

If you look at the implementation of the `system()` function, you will see that it uses `execl()` to execute `/bin/sh`; `execl()` calls `execve()`, passing to it the environment variables array. Therefore, using `system()`, the environment variables of the calling process is passed to the new program `/bin/sh`. Please compile and run the following program to verify this.

2.5 Task 5: Environment Variable and `Set-UID` Programs

`Set-UID` is an important security mechanism in Unix operating systems. When a `Set-UID` program runs, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. `Set-UID` allows us to do many interesting things, but it escalates the user's privilege when executed, making it quite risky. Although the behaviors of `Set-UID` programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how `Set-UID` programs are affected, let us first figure out whether environment variables are inherited by the `Set-UID` program's process from the user's process.

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

Because of the shell program invoked, calling `system()` within a `Set-UID` program is quite dangerous. This is because the actual behavior of the shell program can be affected by environment variables, such as `PATH`; these environment variables are provided by the user, who may be malicious. By changing these variables, malicious users can control the behavior of the `Set-UID` program. In `Bash`, you can change the `PATH` environment variable in the following way (this example adds the directory `/home/seed` to the beginning of the `PATH` environment variable):

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

Although `system()` and `execve()` can both be used to run new programs, `system()` is quite dangerous if used in a privileged program, such as `Set-UID` programs. We have seen how the PATH environment variable affect the behavior of `system()`, because the variable affects how the shell works. `execve()` does not have the problem, because it does not invoke shell. Invoking shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us look at the following scenario.

Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's `Unix` system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run `/bin/cat` to display the specified file. Since the program is running as a root, it can display any file Bob specifies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any file.

二、实验内容及步骤

1. Manipulating Environment Variables

   （1） 使用 printenv 或 env 指令打印出环境变量，也可打印某些特定环境变量（如 PWD 和 DISPLAY）

```
[07/06/21]seed@VM:~$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2537,unix/VM:/tmp/.ICE-unix/2
537
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2489
GTK_MODULES=gail:atk-bridge
PWD=/home/seed
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
HOME=/home/seed
```

```
[07/06/21]seed@VM:~$ printenv PWD
/home/seed
[07/06/21]seed@VM:~$ printenv DISPLAY
:0
[07/06/21]seed@VM:~$
```

（2） 使用 export 和 unset 指令来改变环境变量。实验中以 SET 为例，实现创建和删除一个环境变量。

```
[07/06/21]seed@VM:~$ printenv SET
[07/06/21]seed@VM:~$ export SET=/home/seed/set
[07/06/21]seed@VM:~$ printenv SET
/home/seed/set
[07/06/21]seed@VM:~$ unset SET
[07/06/21]seed@VM:~$ printenv SET
[07/06/21]seed@VM:~$
```

2. Passing Environment Variables from Parent Process to Child Process

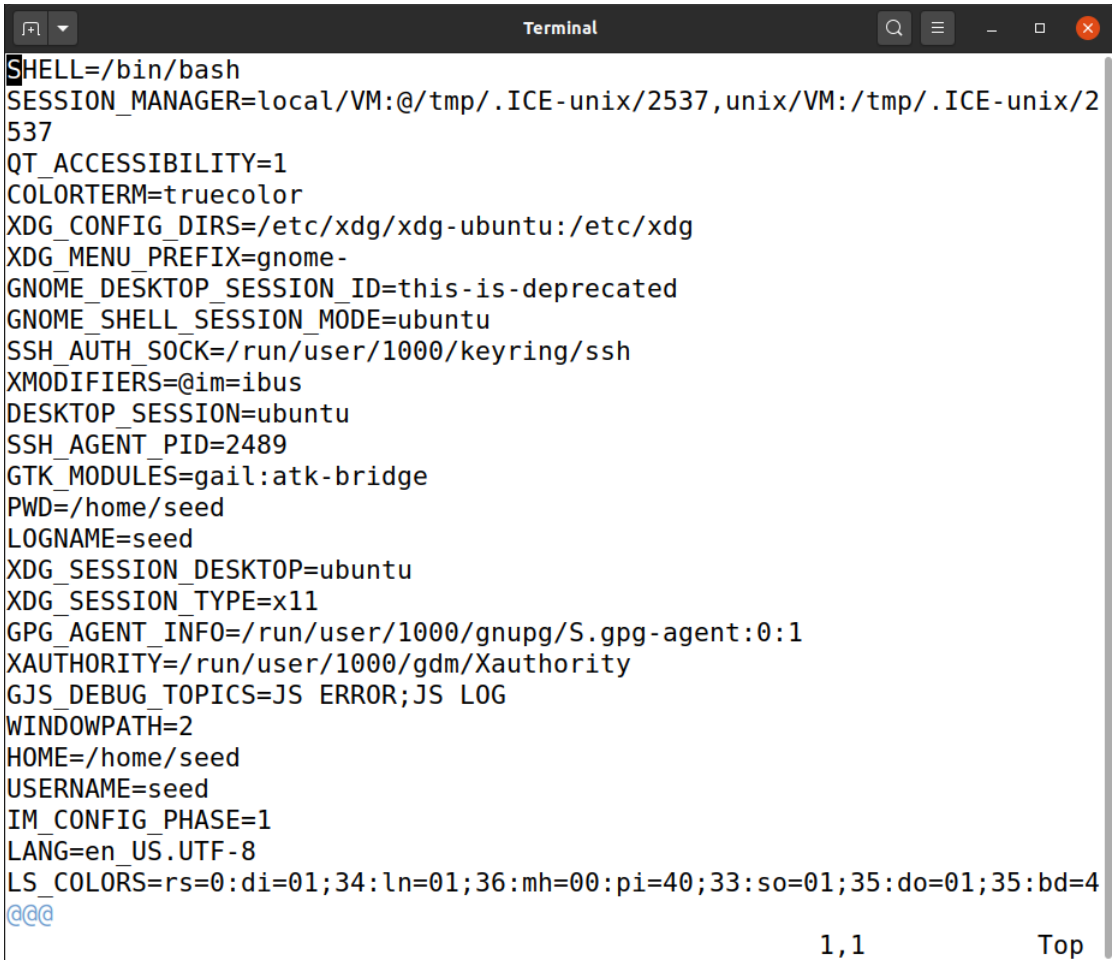
（1）运行如图所示的代码，得到如下结果，即实现了打印环境变量的功能。并将其保存到一个名为 file1 的文件中。

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void printenv()
{
  int i = 0;
  while (environ[i] != NULL) {
     printf("%s\n", environ[i]);
     i++;
  }
}

void main()
{
  pid_t childPid;
  switch(childPid = fork()) {
    case 0:  /* child process */
      printenv();
      exit(0);
    default:  /* parent process */
       //printenv();
      exit(0);
  }
}
```



（2）注释掉语句 1，并将语句 2 解除注释，再将结果打印到 file2 文件中，使用 diff 指令进行比较，发现两个文件相同。

```
[07/06/21]seed@VM:~$ gcc myprintenv.c
[07/06/21]seed@VM:~$ a.out > file1
[07/06/21]seed@VM:~$ gcc myprintenv.c
[07/06/21]seed@VM:~$ a.out > file2
[07/06/21]seed@VM:~$ diff file1 file2
[07/06/21]seed@VM:~$
```

（3）分析：子进程会继承父进程的环境变量。由于子进程继承了父进程的环境变量，且父进程环境变量没有发生改变，故两次输出是相同的。

3. Environmental Variables and execve()

（1） 执行下图所示的程序，结果如图，没有打印出任何全局变量

```c
#include <unistd.h>

extern char **environ;
int main()
{
  char *argv[2];

  argv[0] = "/usr/bin/env";
  argv[1] = NULL;
  execve("/usr/bin/env", argv, NULL);     ①

  return 0 ;
}
```

```
[07/06/21]seed@VM:~/setuid$ gcc myenv.c
[07/06/21]seed@VM:~/setuid$ ./a.out
[07/06/21]seed@VM:~/setuid$
```

（2） 将 execve 函数中的第三个参数由 NULL 改为 environ 后，编译程序，观察到如下结果，成功打印出全局变量。

```
[07/06/21]seed@VM:~/setuid$ gcc myenv.c
[07/06/21]seed@VM:~/setuid$ ./a.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2537,unix/VM:/tmp/.ICE-unix/2
537
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2489
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/setuid
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
```

（3） 结论：根据操作系统有关知识，调用 fork()函数产生的新进程的环境变量继承于父进程，而调用 execve()函数则需要重新赋予，故在输入 NULL 时不打印环境变量；而经过修改后，传递了外部环境变量数据给新程序，从而打印出了当前环境变量。

4. Environmental Variables and system()

编译如下代码并得到结果

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  system("/usr/bin/env");
  return 0 ;
}
```

```
[07/06/21]seed@VM:~/setuid$ touch task4.c
[07/06/21]seed@VM:~/setuid$ gcc task4.c
[07/06/21]seed@VM:~/setuid$ ./a.out
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
SSH_AGENT_PID=2489
XDG_SESSION_TYPE=x11
SHLVL=1
HOME=/home/seed
OLDPWD=/home/seed
DESKTOP_SESSION=ubuntu
GNOME_SHELL_SESSION_MODE=ubuntu
GTK_MODULES=gail:atk-bridge
MANAGERPID=2282
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
COLORTERM=truecolor
IM_CONFIG_PHASE=1
LOGNAME=seed
JOURNAL_STREAM=9:35488
_=./a.out
```

5. Environmental Variables and Set-UID Programs

（1） 编译下图所示的程序

```c
#include <stdio.h>
#include <stdlib.h>

extern char **environ;
int main()
{
  int i = 0;
  while (environ[i] != NULL) {
    printf("%s\n", environ[i]);
    i++;
  }
}
```

```
[07/06/21]seed@VM:~/setuid$ touch task5.c
[07/06/21]seed@VM:~/setuid$ gcc task5.c -o  task5
[07/06/21]seed@VM:~/setuid$ ./task5
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2537,unix/VM:/tmp/.ICE-unix/2
537
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2489
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/setuid
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
LANG=en_US.UTF-8
```

（2） 输入以下命令改变该程序的权限，再修改 PATH、LD_LIBRARY_PATH、WHAT 环境变量的值（WHAT 是由自己定义的）

```
[07/06/21]seed@VM:~/setuid$ sudo chown root task5
[07/06/21]seed@VM:~/setuid$ sudo chmod 4755 task5
```

```
[07/06/21]seed@VM:~/setuid$ export WHAT=$WHAT:/home/what
[07/06/21]seed@VM:~/setuid$ printenv WHAT
:/home/what
[07/06/21]seed@VM:~/setuid$ export PATH=$PATH:/home/path
[07/06/21]seed@VM:~/setuid$ printenv PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/game
s:/usr/local/games:/snap/bin:.:/home/path
[07/06/21]seed@VM:~/setuid$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH=/h
ome/ld
[07/06/21]seed@VM:~/setuid$
```

（3） 再执行（1）中给出的程序，可以观察到此时的环境变量 PATH、LD_LIBRARY_PATH、WHAT 的值已经是修改后的值。注意到 LD_LIBRARY_PATH 首次没有打印出来，我们通过给 setuid 程序降权，再次运行可观察到 LD_LIBRARY_PATH。

```
[07/06/21]seed@VM:~/setuid$ sudo chmod 777 task5
[07/06/21]seed@VM:~/setuid$ sudo chown seed task5
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr
/games:/usr/local/games:/snap/bin:.:/home/path
WHAT=:/home/what
LD_LIBRARY_PATH==/home/ld
```

（4）分析：根据操作系统有关知识，shell 的子进程会继承 shell 中设置的环境变量，且这种继承是通过 fork()函数实现的。这里可以认为是从两个方向说明了当前运行的是 setuid 程序，正向是 export 的环境变量的输出，反向是 LD_LIBRARY_PATH 变量的忽略。

6. The PATH Environment Variable and Set-UID Programs

（1）根据下图代码，并将其改为 Set-UID 程序后进行编译得到如下结果（与/bin/ls 命令得到的结果相同）：

```c
int main()
{
  system("ls");
  return 0;
}
```

```
[07/06/21]seed@VM:~/setuid$ touch task6.c
[07/06/21]seed@VM:~/setuid$ /bin/ls
a.out       catall.c  myenv.c       task4.c  task5.c
cap_leak.c  child.c   myprintenv.c  task5    task6.c
[07/06/21]seed@VM:~/setuid$ gcc task6.c -o task6
task6.c: In function ‘main’:
task6.c:3:1: warning: implicit declaration of function ‘system’ [-Wimp
licit-function-declaration]
    3 | system("ls");
      | ^~~~~~
[07/06/21]seed@VM:~/setuid$ ./task6
a.out       catall.c  myenv.c       task4.c  task5.c  task6.c
cap_leak.c  child.c   myprintenv.c  task5    task6
[07/06/21]seed@VM:~/setuid$
```

（2）将其改为 UID program 后，运行程序，发现没有区别

```
[07/06/21]seed@VM:~/setuid$ sudo chown root task6
[07/06/21]seed@VM:~/setuid$ sudo chmod 4755 task6
[07/06/21]seed@VM:~/setuid$ ./task6
a.out       catall.c  myenv.c       task4.c  task5.c  task6.c
cap_leak.c  child.c   myprintenv.c  task5    task6
[07/06/21]seed@VM:~/setuid$
```

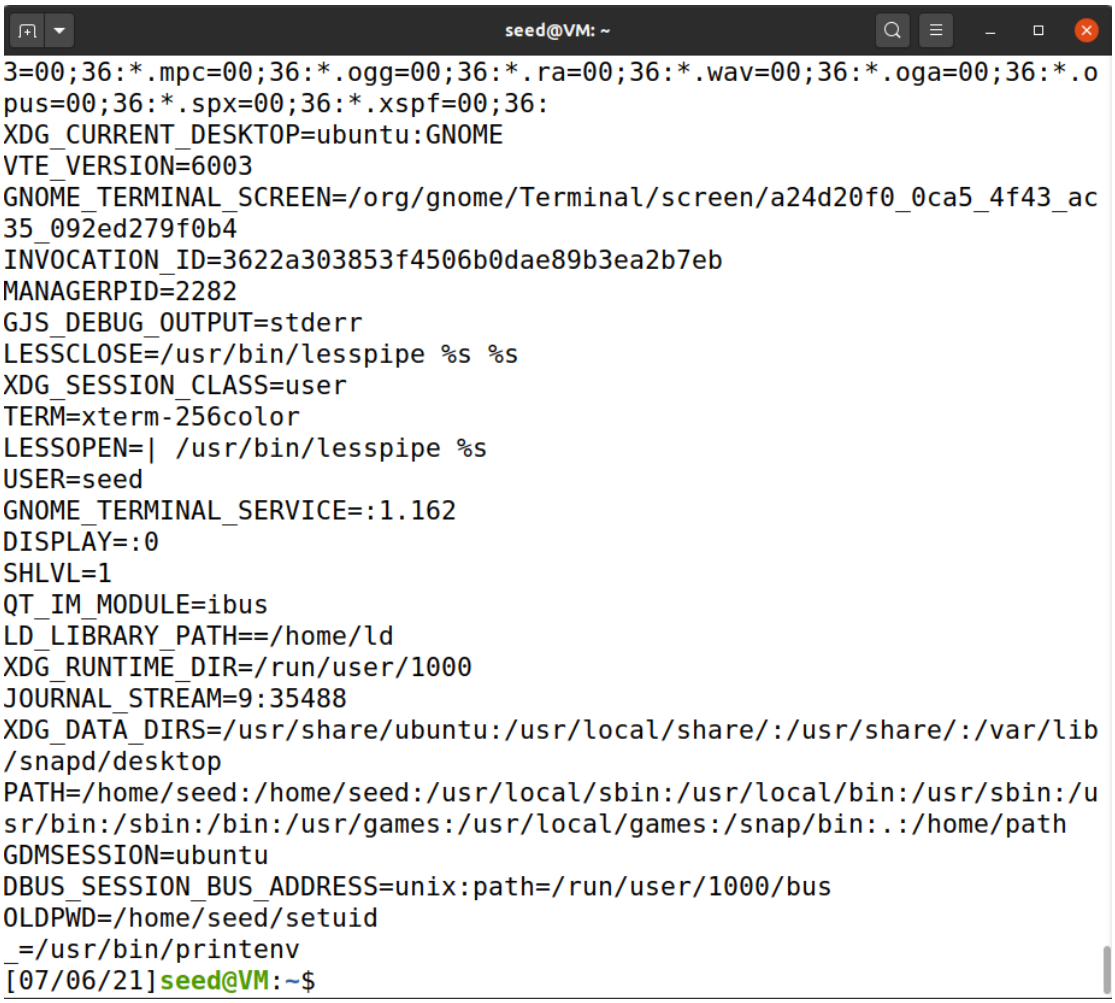
（3）将当前目录添加到 PATH，接下来编写一个 badcode 程序，且将编译生成的可执行文件命名为 ls

```
[07/06/21]seed@VM:~$ export PATH=/home/seed:$PATH
```

Open　badcode.c　Save

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
extern char **environ;
int main()
{
        char *argv[2];
        argv[0]="/usr/bin/env";
        argv[1]=NULL;
        execve("usr/bin/env",argv,environ);
        return 0;
}
```

（4）再次运行之前的 task6



（5）分析：此时程序并没有像之前一样运行/bin/ls，而是运行当前目录下的 ls，而 ls 被定义为 execve 执行/usr/bin/env，所以可以看到结果是打印出了环境变量。原因是 task6 中 system()函数的命令没有绝对路径，于是会在环境变量中寻找含有 ls 的目录，所以由于我们通过在 PATH 中加入当前目录，使得链接器在寻找环境变量时首先找到当前目录，而不是/bin/ls，于是便生成了上述结果。事实上，本例中的 badcode 只是执行了/usr/bin/env，但 badcode 的内容可以由攻击者自己决定，因此可能产生非常大的安全漏洞，如让攻击者得到用户密码。

8. Invoking External Programs Using system() versus execve()

（1）编译如图所示的代码，并将其设置为 UID 程序

Open ▾ | catall.c ~/setuid | Save

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char *v[3];
  char *command;

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;

  command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
  sprintf(command, "%s %s", v[0], v[1]);

  // Use only one of the followings.
  system(command);
  // execve(v[0], v, NULL);

  return 0 ;
}
```

（2）为了更好地观察本实验的效果，在工作目录下添加两个文件 test1 和 test2，设为 UID 程序。

（3）运行程序，先后执行修改 test1 文件名（改为 hello）和删除文件 test2 的操作。可以发现修改文件名和删除文件都是成功的。

```
[07/06/21]seed@VM:~$ cd /home/seed/setuid/task8
[07/06/21]seed@VM:~/.../task8$ ls
catall.c  test1  test2
[07/06/21]seed@VM:~/.../task8$ sudo su
root@VM:/home/seed/setuid/task8#
root@VM:/home/seed/setuid/task8# gcc -o demo8 catall.c
root@VM:/home/seed/setuid/task8# chown root demo8
root@VM:/home/seed/setuid/task8# chmod 4755 demo8
root@VM:/home/seed/setuid/task8# ls
catall.c  demo8  test1  test2
root@VM:/home/seed/setuid/task8# chown root test1
root@VM:/home/seed/setuid/task8# chmod 4755 test1
root@VM:/home/seed/setuid/task8# chown root test2
root@VM:/home/seed/setuid/task8# chmod 4755 test2
root@VM:/home/seed/setuid/task8# exit
exit
[07/06/21]seed@VM:~/.../task8$ demo8 "test1:mv test1 hello"
```

```
/bin/cat: 'test1:mv': No such file or directory
/bin/cat: hello: No such file or directory
[07/06/21]seed@VM:~/.../task8$ test1;mv test1 hello
[07/06/21]seed@VM:~/.../task8$
[07/06/21]seed@VM:~/.../task8$
[07/06/21]seed@VM:~/.../task8$ ls
catall.c  demo8  hello  test2
[07/06/21]seed@VM:~/.../task8$ demo8 "test2;rm test2"
rm: remove write-protected regular empty file 'test2'? y
[07/06/21]seed@VM:~/.../task8$ ls
catall.c  demo8  hello
[07/06/21]seed@VM:~/.../task8$
```

（4） 注释掉 system()语句，解除 execve()语句的注释。运行截图如下，发现改名和删除操作均不成功。

```
[07/06/21]seed@VM:~/.../task8$ gcc catall.c -o demo8
[07/06/21]seed@VM:~/.../task8$ sudo su
root@VM:/home/seed/setuid/task8# chown root demo8
root@VM:/home/seed/setuid/task8# chmod 4755 demo8
root@VM:/home/seed/setuid/task8# ls
catall.c  demo8  hello
root@VM:/home/seed/setuid/task8# exit
exit
[07/06/21]seed@VM:~/.../task8$ demo8 "hello;mv hello hi"
/bin/cat: 'hello;mv hello hi': No such file or directory
[07/06/21]seed@VM:~/.../task8$ demo8 "hello;rm hello"
/bin/cat: 'hello;rm hello': No such file or directory
[07/06/21]seed@VM:~/.../task8$ ls
catall.c  demo8  hello
[07/06/21]seed@VM:~/.../task8$
```

（5） 分析：在修改程序之后，发现第一次的攻击方式便不再成功了。结合前面实验的经验，猜想可能是因为本实验中执行 execve()语句时，未传入环境变量参数，导致指令不能被正常执行。