Environment Variable and Set-UID Program Lab Report

Mudit Vats
mpvats@syr.edu
10/06/2018

Table of Contents

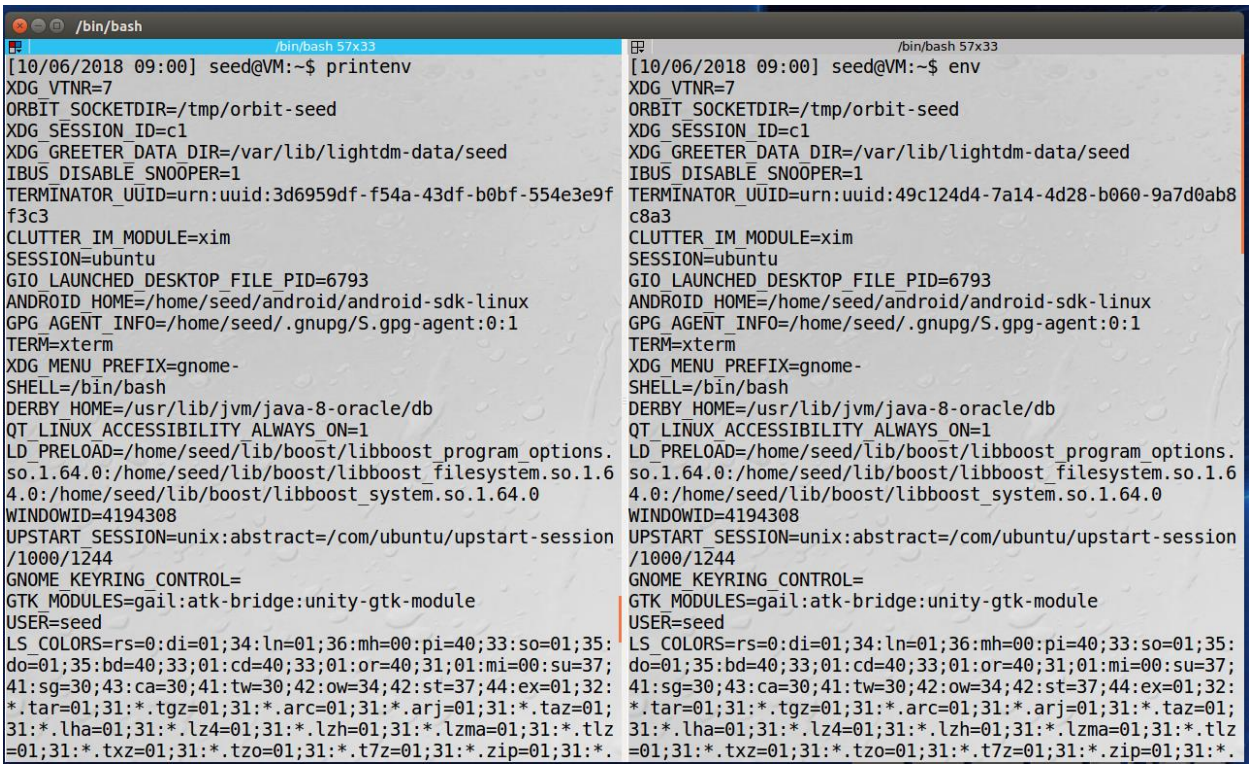
Overview

This lab report presents observations and explanations for the tasks described in the Environment Variable and Set-UID Program Lab.

Please note, I updated the prompt in the .bashrc file to display time. This prompt configuration already existed in the .bashrc file, but it was commented out. I uncommented per the Lab Report Requirements to provide a timestamp for all screen captures.

Task 1: Manipulating Environment Variables

Using printenv (left) and env (right) to output environment variables.

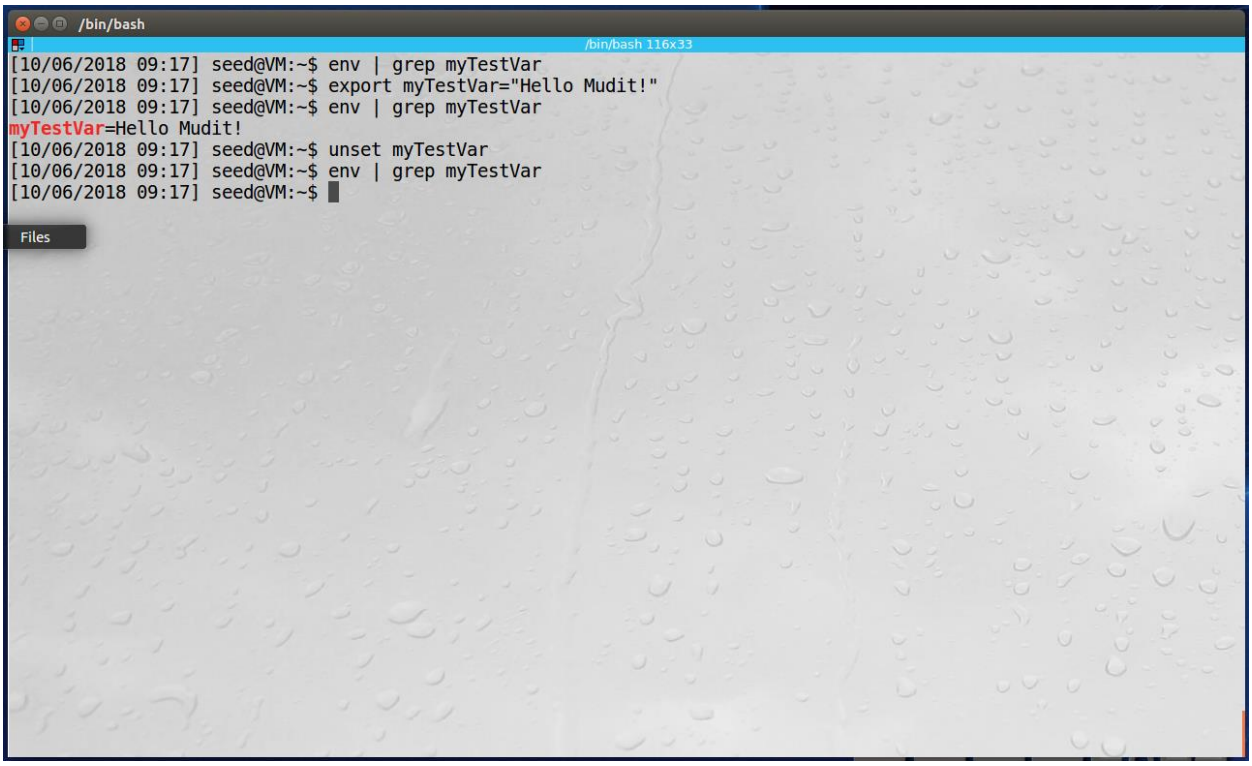


Observations / Explanation

Both commands display all environment variables defined. They display the same list of variable and values – I scrolled down and compared the output of both and they were identical. Only difference was the last environment variable, ("_=") which is assigned to the current executable being executed (e.g. "_=/usr/bin/printenv" or "_=/usr/bin/env"). From the perspective of outputting environment variables, they appear to be functionally equivalent.

Use export and unset

Example using export and unset below.

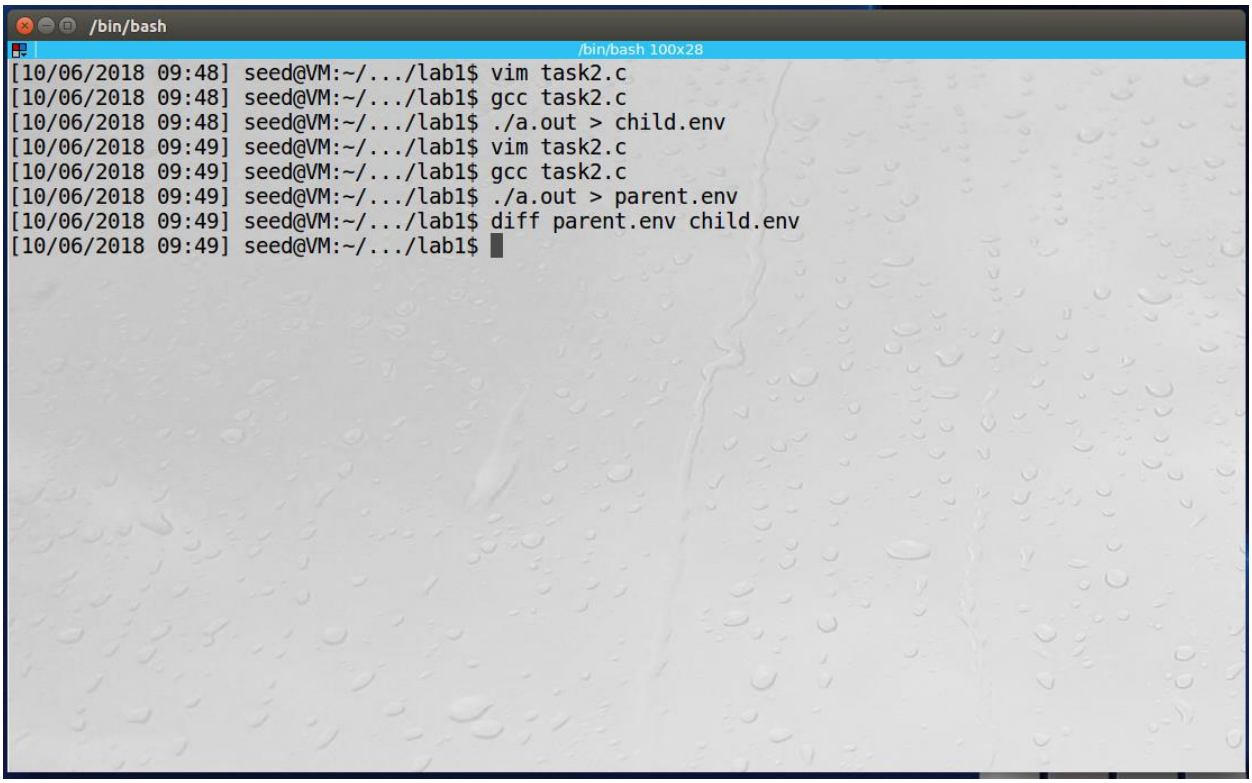


Observations / Explanation

In this example, we can see that myTestVar is not defined as an environment variable. The variable is defined as "Hello Mudit!" via the export command. Checking again, we can now see that the environment variable myTestVar is now defined. We can then unset it using the unset myTestVar command. By using export and unset, we can set and unset variables in the current shell environment.

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

Using the program described in Task 2.2 Step 1, I compiled the program which, without modification, outputs the environment variables for the child process.



Observations / Explanations

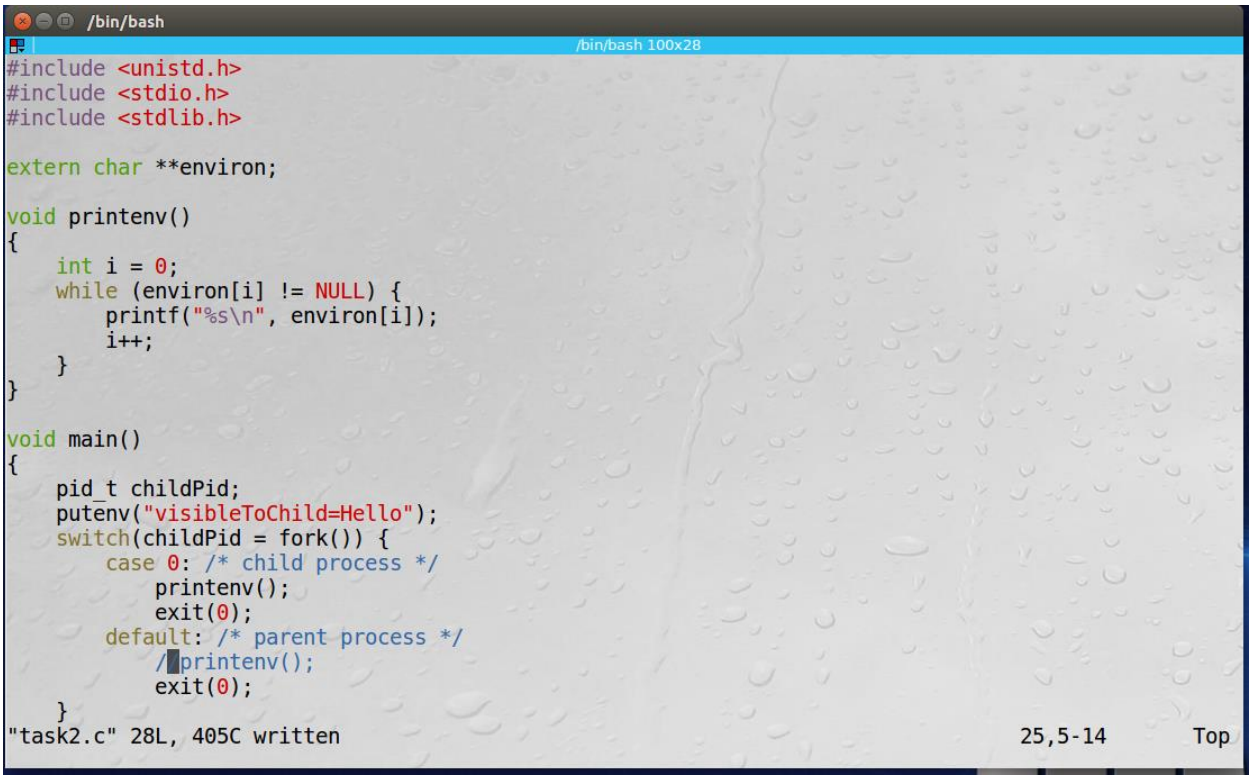
I compiled the program (task2.c) first, which, as mentioned, outputs the environment variables for the child process. I executed in the line "./a.out > child.env". child.env contains the environment variables. Viewing the file, I can see the environment variables which are inherited by the child process.

I then modified task2.c to printenv() of the parent process. I output the environment variables to parent.env. Viewing the file, I can see the environment variables inherited by the parent process from the shell environment.

Finally, I diff the child.env and parent.env and see that there are no differences. The files' contents and the file sizes are identical. The conclusion is that the child process inherits the same environment variables as the parent.

Variable Manipulation (more investigation)

Since this is a Computer Security class, I investigated whether we can manipulate environment variables prior to forking the child process. I added the putenv("VisibleToChild=Hello") prior to forking the child process.

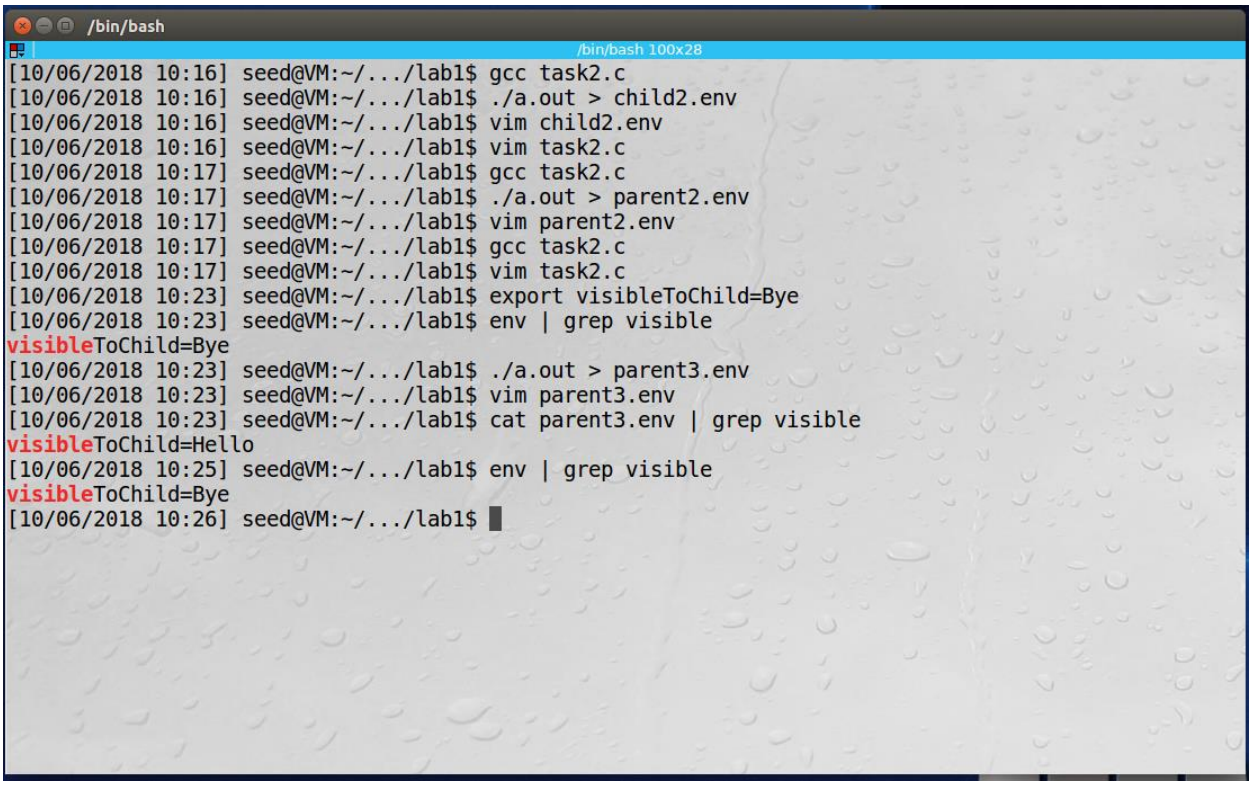


Observations / Explanation

By adding the variable, visibleToChild, prior to forking, the application was able to add an environment variable which then became visible in the child process.

I commented the child printenv and uncommented the parent printenv. In this case as well, the visibleToChild variable was visible since the putenv was set prior to the fork so both the parent and subsequent fork contain the environment variable.

Further, I defined "visibleToChild" before executing a.out and observed the behavior. In this case, the application modified the environment variable for the process. It did not change the shell's environment variable, just the definition of the variable inside the process (important distinction!).



The point being that while a process does inherit the environment variables which are defined in the shell environment, applications can add and manipulate variables programmatically and present something different to forked process.

Task 3: Environment Variables and execve()

Execute the following program NULL as the envp argument and environ as the envp argument. Observe results.

```
/bin/bash
/bin/bash 109x28
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.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, NULL);
    //execve("/usr/bin/env", argv, environ);

    return 0 ;
}
~
~
~
~
~
~
~
~
"task3.c" 19L, 273C written                                   1,1           All
```

Observations and Explanation

When we execute the first time, no environment variables are outputted.

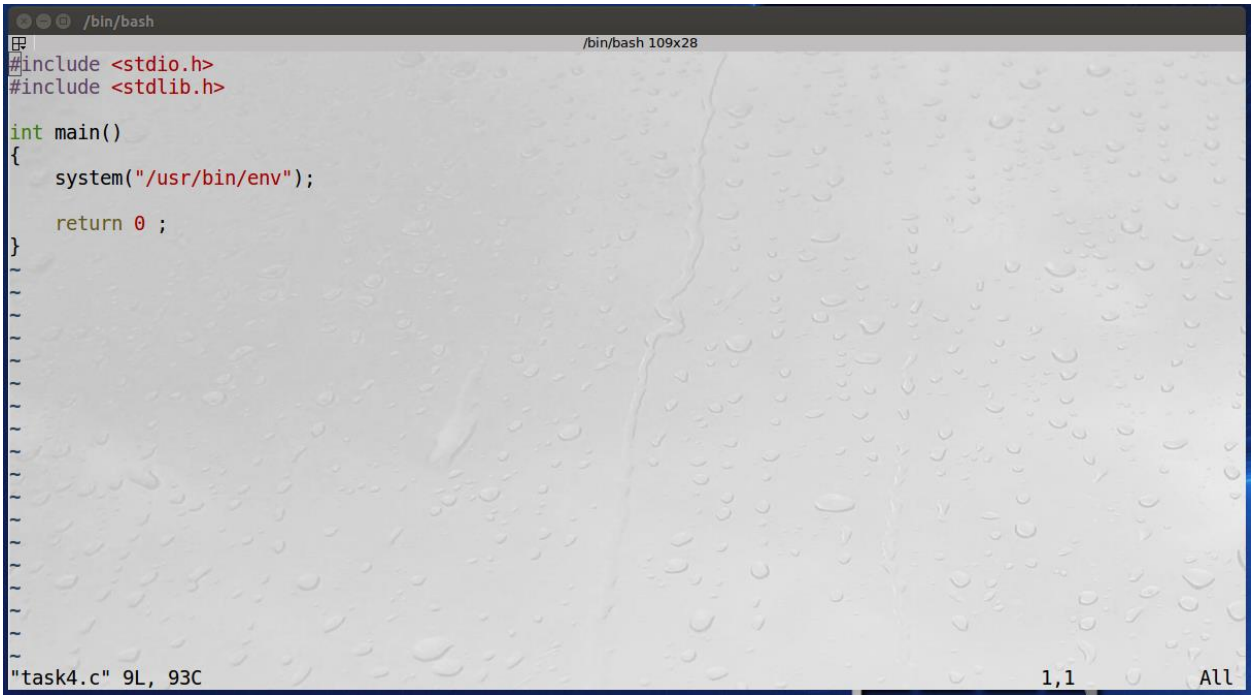
When we executed the second time, with the environ being passed in, all environment variables are outputted.

```
/bin/bash
/bin/bash 109x28
[10/06/2018 10:58] seed@VM:~/.../lab1$ gcc task3.c
[10/06/2018 10:58] seed@VM:~/.../lab1$ ./a.out
[10/06/2018 10:58] seed@VM:~/.../lab1$ vim task3.c
[10/06/2018 10:58] seed@VM:~/.../lab1$ gcc task3.c
[10/06/2018 10:58] seed@VM:~/.../lab1$ ./a.out
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
TERMINATOR_UUID=urn:uuid:09e61dc9-af97-45ae-ae15-f8313d458e62
IBUS_DISABLE_SNOOPER=1
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE_PID=3064
ANDROID_HOME=/home/seed/android/android-sdk-linux
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
LD_PRELOAD=/home/seed/lib/boost/libboost_program_options.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.s
o.1.64.0:/home/seed/lib/boost/libboost_system.so.1.64.0
WINDOWID=23068676
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1368
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
USER=seed
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:s
```

We can see that using a NULL value for the envp parameter implies zero environment variables therefore no environment variables are printed out. By passing in "environ", which points to the environment variables, we can see the environment variables are outputted. The interesting observation is that with execve() we can control the environment variables which are made available to the executing program (first argument of execve()). From a security perspective, this is very important since we can create a predictable state of execution for a given program.

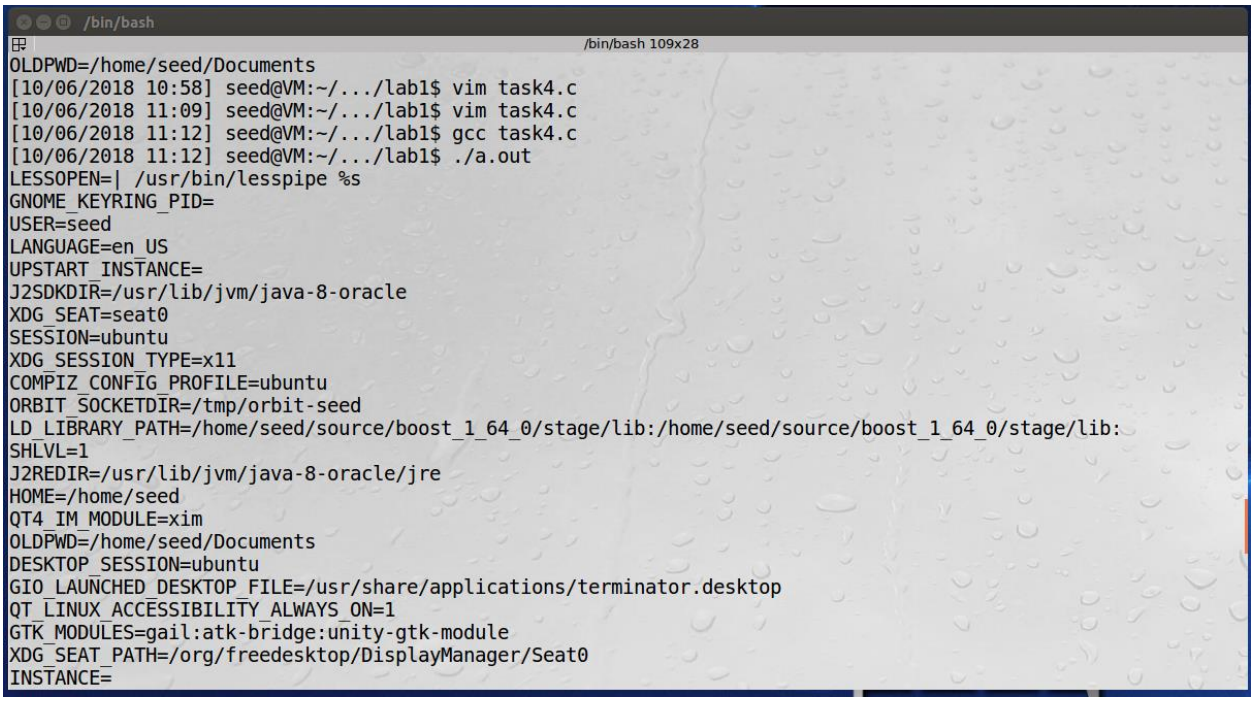
Task 4: Environment Variables and system()

Execute the system() command and observe environment variables.



Observations and Explanation

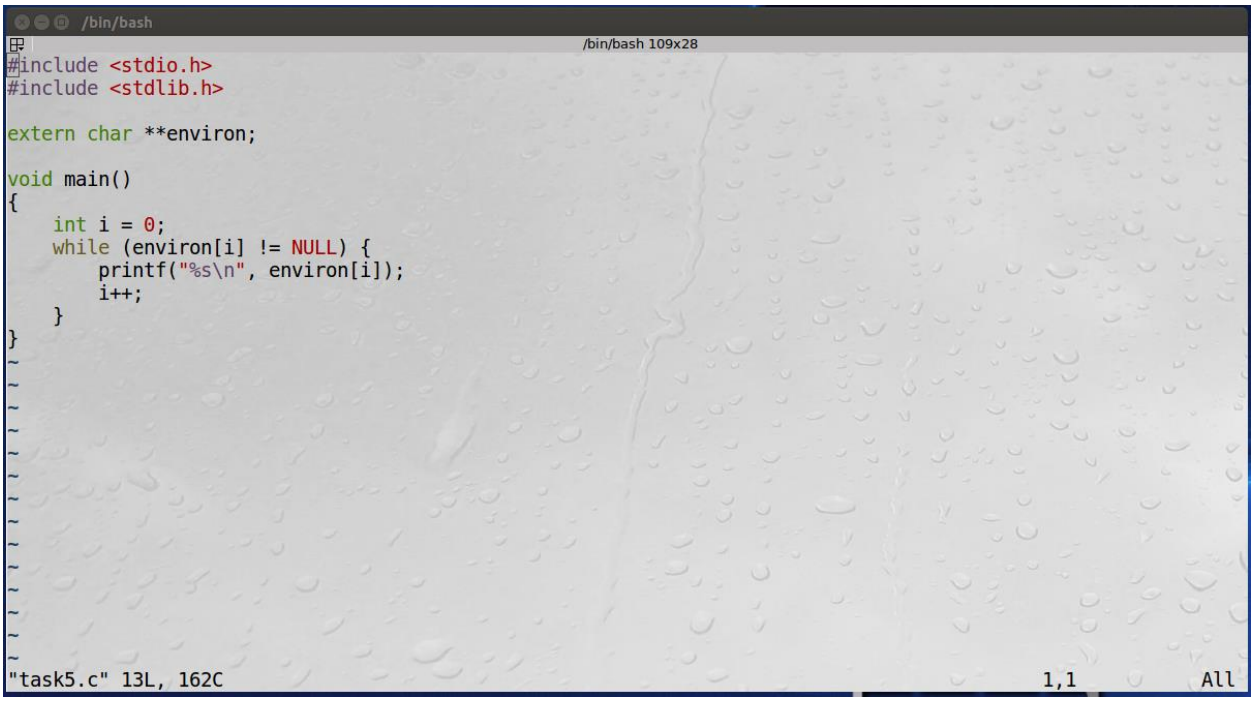
Using system() to execute /usr/bin/env, we do see the environments variables outputted. Since system ultimately uses execve(), we can observe that the environment variables are being passed to execve() and subsequently outputted via the env executable.



Seems like using system() is an easier way to execute executables from a C program, but, since the environment variables are taken as-is from the shell environment, this could introduce potential security risks. This can be mitigated by explicitly using execve().

Task 5: Environment Variable and Set-UID Programs

Understand how environment variables affect Set-UID programs.



We compile the program, task5.c, into "foo". We then set the owner to root and give it set-uid privalages. We can verify that set-uid and the owner is set correctly in the "ls -al" line. Here we can see that foo is owned by "root" and does indeed have the set-uid "s" bit set. This means that when foo is executed, it will execute as root and with root privileges.

We also define MUDITVAR per the exercise. We do not redefine PATH and LD_LIBRARY_PATH since those are already defined in the environment.

```
/bin/bash
/bin/bash 109x28
[10/06/2018 11:29] seed@VM:~/.../lab1$ vim task5.c
[10/06/2018 11:29] seed@VM:~/.../lab1$ gcc task5.c
[10/06/2018 11:29] seed@VM:~/.../lab1$ vim task5.c
[10/06/2018 11:31] seed@VM:~/.../lab1$ gcc task5.c
[10/06/2018 11:31] seed@VM:~/.../lab1$ gcc task5.c -o foo
[10/06/2018 11:31] seed@VM:~/.../lab1$ sudo chown root foo
[sudo] password for seed:
[10/06/2018 11:31] seed@VM:~/.../lab1$ sudo chmod 4755 foo
[10/06/2018 11:31] seed@VM:~/.../lab1$ ls -al foo
-rwsr-xr-x 1 root seed 7396 Oct  6 11:31 foo
[10/06/2018 11:32] seed@VM:~/.../lab1$ env | grep PATH
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:
.:/snap/bin:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/b
in:/home/seed/android/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/home/seed/
android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_BIN_PATH=/usr/bin/
[10/06/2018 11:32] seed@VM:~/.../lab1$ env | grep LD_LIBRARY_PATH
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
[10/06/2018 11:33] seed@VM:~/.../lab1$ export MUDITVAR=hello
[10/06/2018 11:33] seed@VM:~/.../lab1$ env | grep MUDIT
MUDITVAR=hello
[10/06/2018 11:33] seed@VM:~/.../lab1$
```

Observations and Explanation

```
/bin/bash
/bin/bash 109x28
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:
.:/snap/bin:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/b
in:/home/seed/android/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/home/seed/
android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_BIN_PATH=/usr/bin/
[10/06/2018 11:32] seed@VM:~/.../lab1$ env | grep LD_LIBRARY_PATH
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
[10/06/2018 11:33] seed@VM:~/.../lab1$ export MUDITVAR=hello
[10/06/2018 11:33] seed@VM:~/.../lab1$ env | grep MUDIT
MUDITVAR=hello
[10/06/2018 11:33] seed@VM:~/.../lab1$ ./foo > foo.out
[10/06/2018 11:44] seed@VM:~/.../lab1$ cat foo.out | grep MUDIT
MUDITVAR=hello
[10/06/2018 11:44] seed@VM:~/.../lab1$ cat foo.out | grep PATH
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:
.:/snap/bin:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/b
in:/home/seed/android/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/home/seed/
android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_BIN_PATH=/usr/bin/
[10/06/2018 11:45] seed@VM:~/.../lab1$ cat foo.out | grep LD_LIBRARY_PATH
[10/06/2018 11:46] seed@VM:~/.../lab1$
```

Interesting observations after executing "./foo > foo.out" –

1. MUDITVAR is defined in the child process. The value is the same.
2. PATH is defined in the child process. The value is the same, including the home directory which is "seed". ← **Surprised by this!**
3. LD_LOAD_LIBRARY is not defined. ← **Surprised by this!**

While we are executing a set-uid program as another user, we are still using the real user id's environment. That is why MUDITVAR and PATH are defined per seed user.

LD_LIBRARY_PATH is used for specifying shared objects (libraries) for dynamically linking with C/CPP applications. It seems that safeguards are in place to ensure that the execution of applications from a set-uid program does not inherit the LD_LIBRARY_PATH. Without this safeguard, a malicious user can put to his/her own shared objects which then the child process's application would link against. This can be hazardous since the malicious user can insert whatever code they desire into the shared objects those causing a security risk.

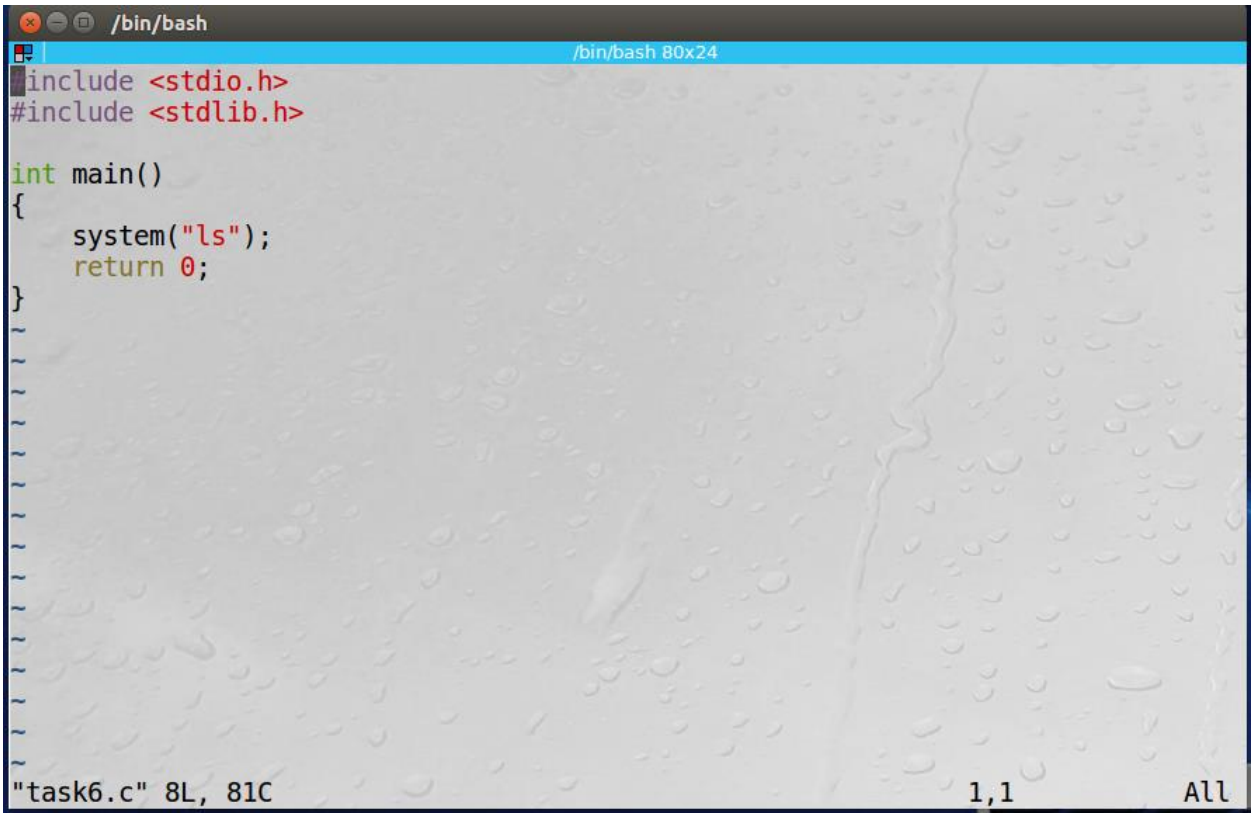
Observations and Explanations (more investigation)

To understand whether the LD_LIBRARY_PATH safeguard is only for root set-uid applications or any set-uid applications, I did not change the foo owner to root, but did set the set-uid bit so that anyone who executes foo takes on seed's credentials.

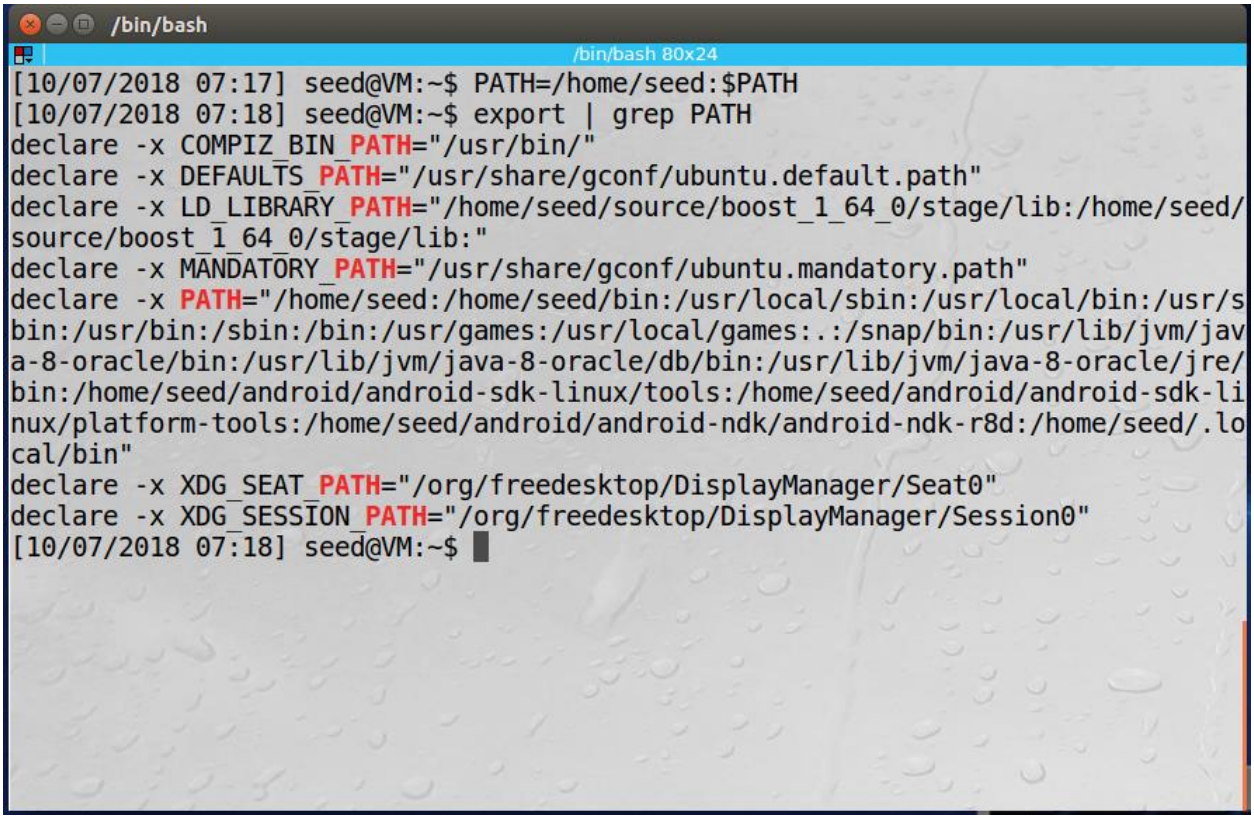
I then logged in as the other user and executed ./foo. I observed the same behavior. LD_LIBRARY_PATH is not inherited by foo. This leads me to conclude that the safeguard for LD_LIBRARY_PATH is per set-uid enabled applications, irrespective of who the owner is.

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

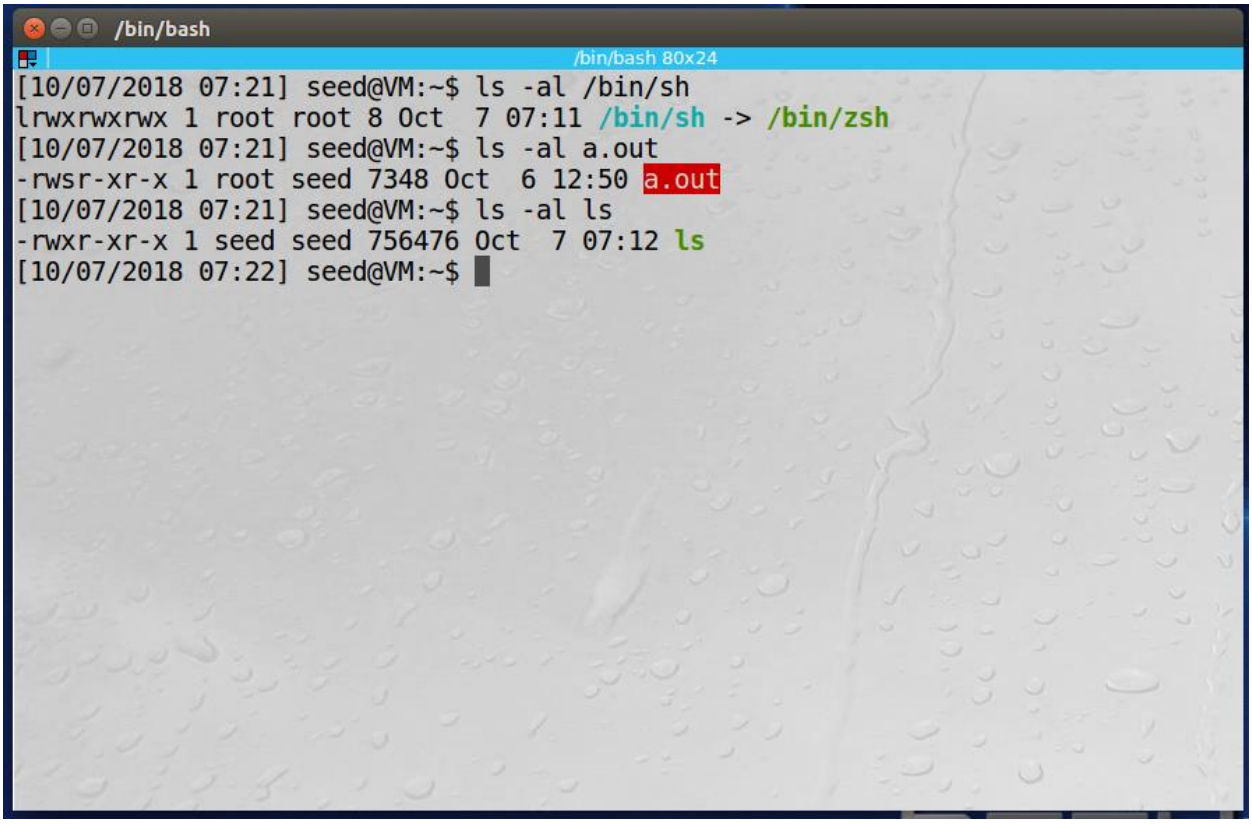
Execute a Set-UID program with a modified PATH. Program and updated PATH to point to the seed home directory shown below. Also, my "ls" program is the zsh shell program as shown below.



Here we set the first directory in the PATH to seed's home directory.

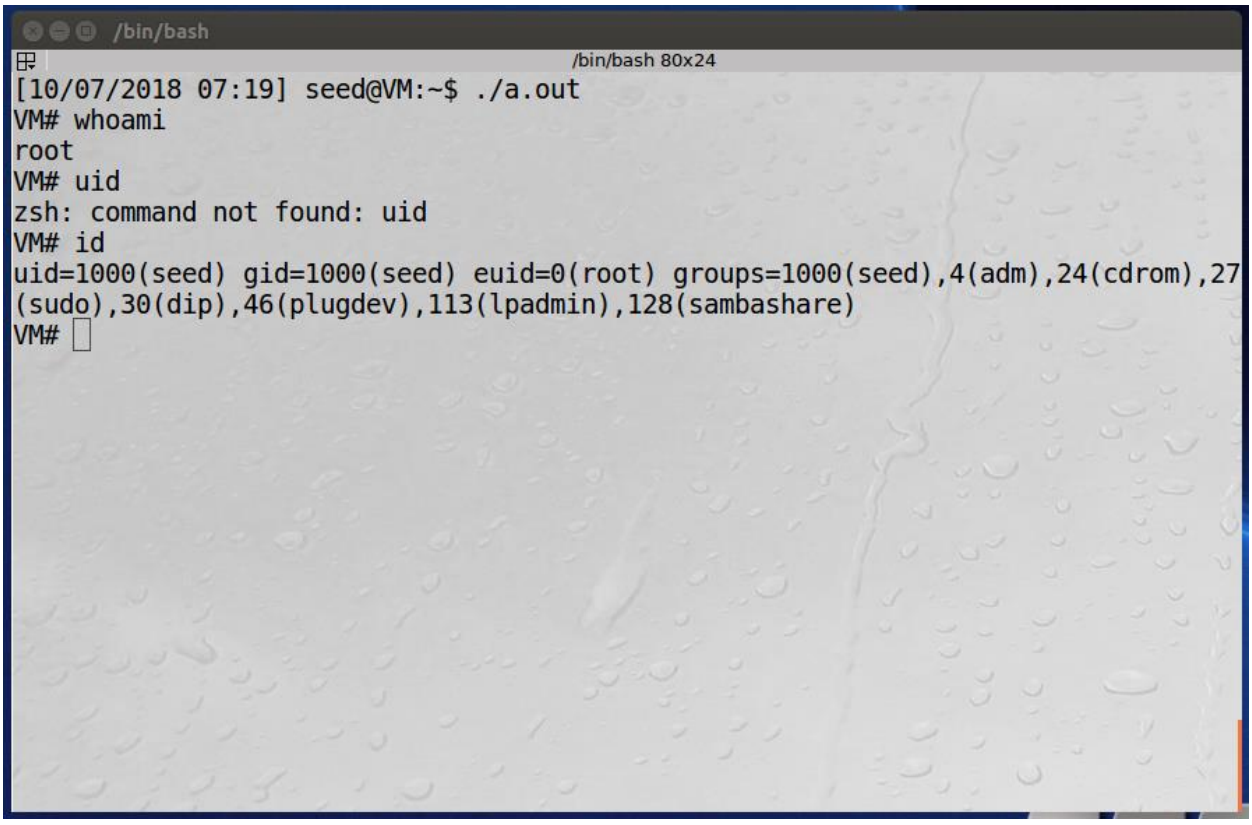


Here we show that we are now point the sh to /bin/zsh, per Task 6 instruction for Ubuntu 16.04. Also showing our ls program (which is actually zsh as mentioned earlier).



Observations and Explanations

My ls program is the zsh shell. So, when ls gets executed, the system() invokes the /bin/sh, which points to zsh per the lab instructions since zsh does not have the vulnerability work-around. When zsh is invoked to call ls, *my* ls runs which then executes the zsh shell that I copied to my home directory.



The result is that, by modifying the PATH variable, I was easily able to get elevated root privileges on the system. As can be seen in the above picture, the real uid is seed which is what I am logged in as, but the effective uid is root, which gives me complete root access.

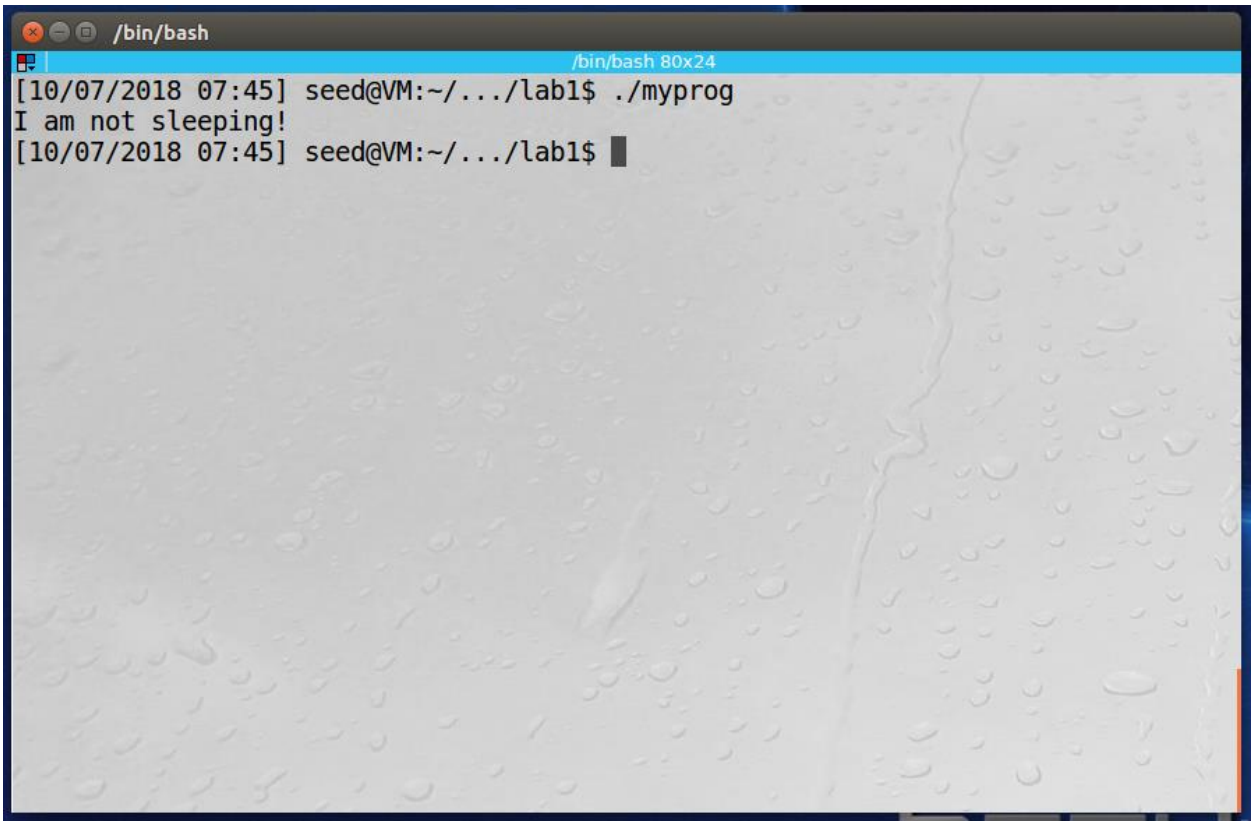
Task 7: The LD_PRELOAD Environment Variable and Set-UID Programs

Create a program and dynamically linked library and test various execution scenarios.

Observations

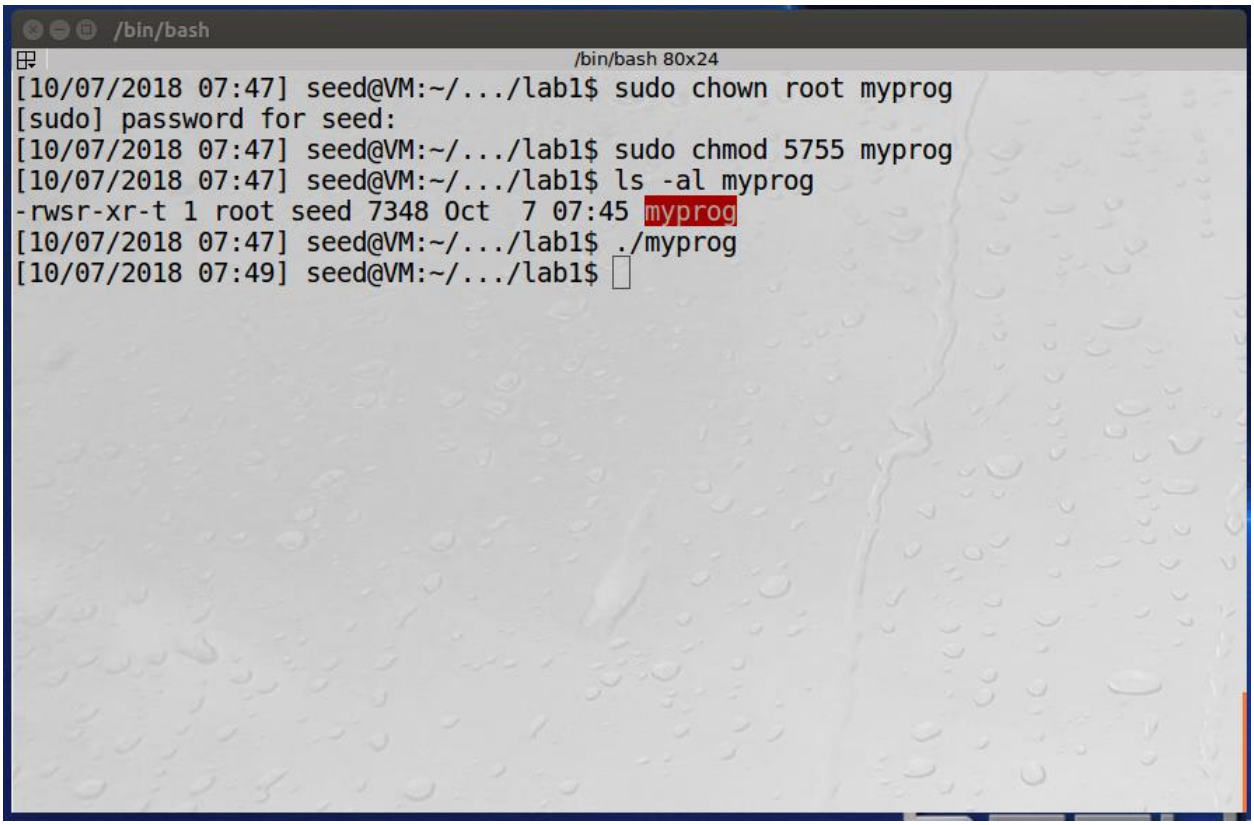
I tested myprog under the following scenarios. Observations for each listed below.

Scenario 1: Make myprog a regular program, and run it as a normal user.



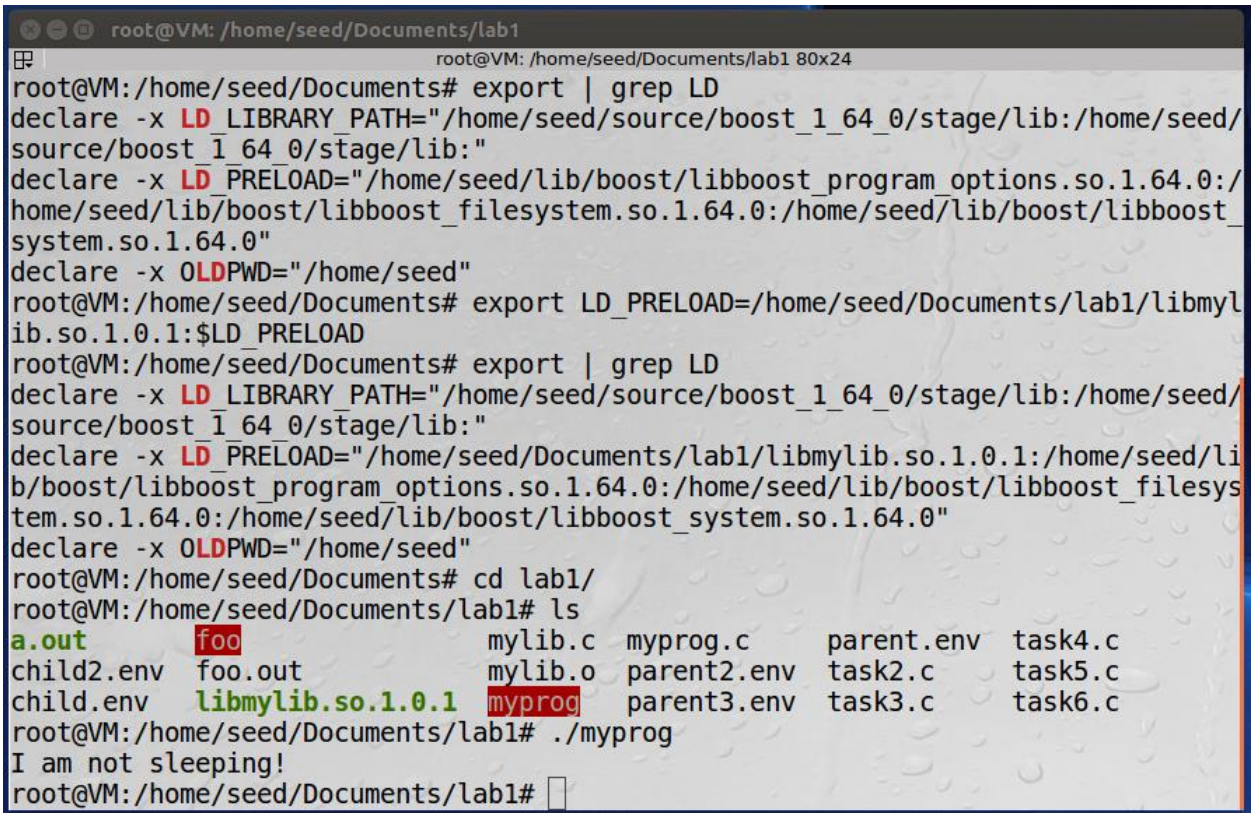
In this scenario, the program myprog linked with the library defined in LD_PRELOAD. We know this since it print out the "I am not sleeping!" string.

Scenario 2: Make myprog a Set-UID root program, and run it as a normal user.



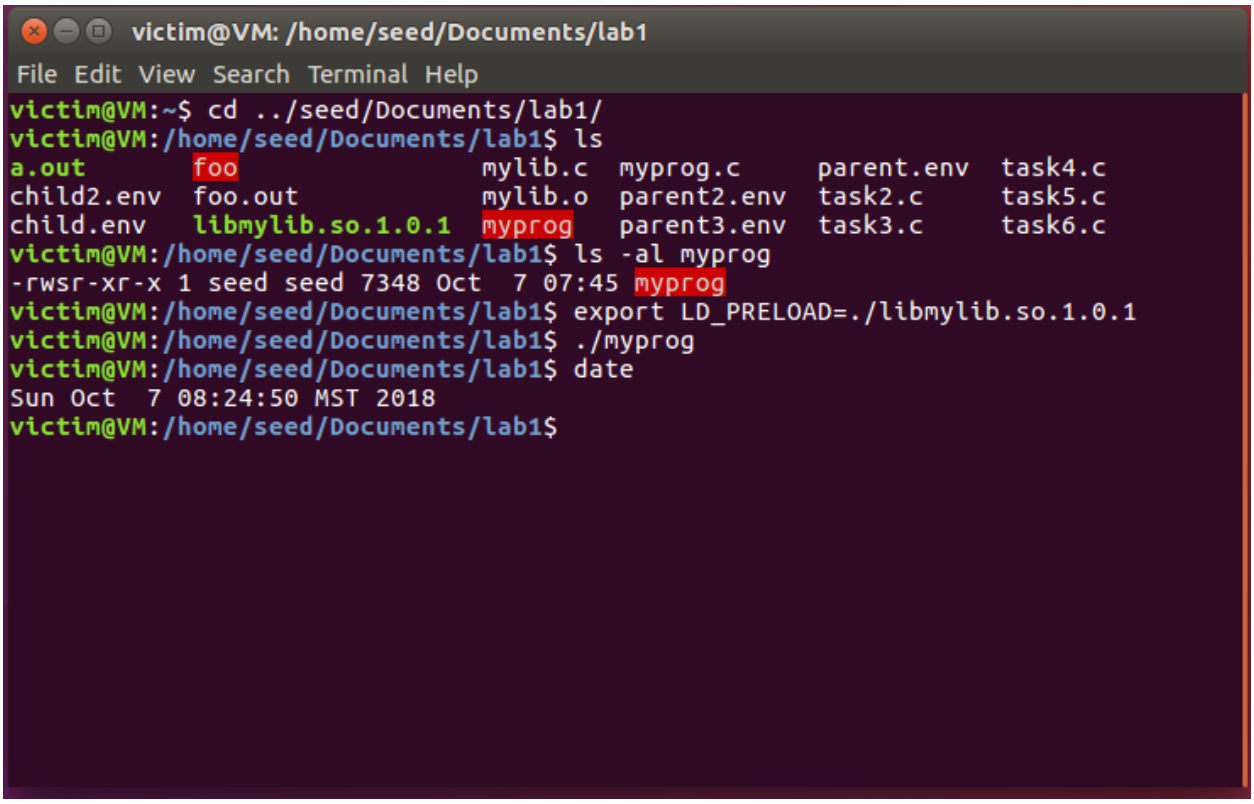
In this scenario, the program did not print the "I am not sleeping!" message, so it did not link with the dynamically linked library we created. LD_PRELOAD may not have been inherited here.

Scenario 3: Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.



In this scenario, logged in as root and setting LD_PRELOAD to point to the dynamically linked library we created, executing myprog yielded the "I am not sleeping!" result.

Scenario 4: Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user's account (not-root user) and run it.



In this scenario, we set the LD_PRELOAD and executed myprog from the victim account. The myprog application, however, is owned by seed. In this case, it did not use the LD_PRELOAD and load our dynamically linked library. It used the systems sleep function.

Explanation

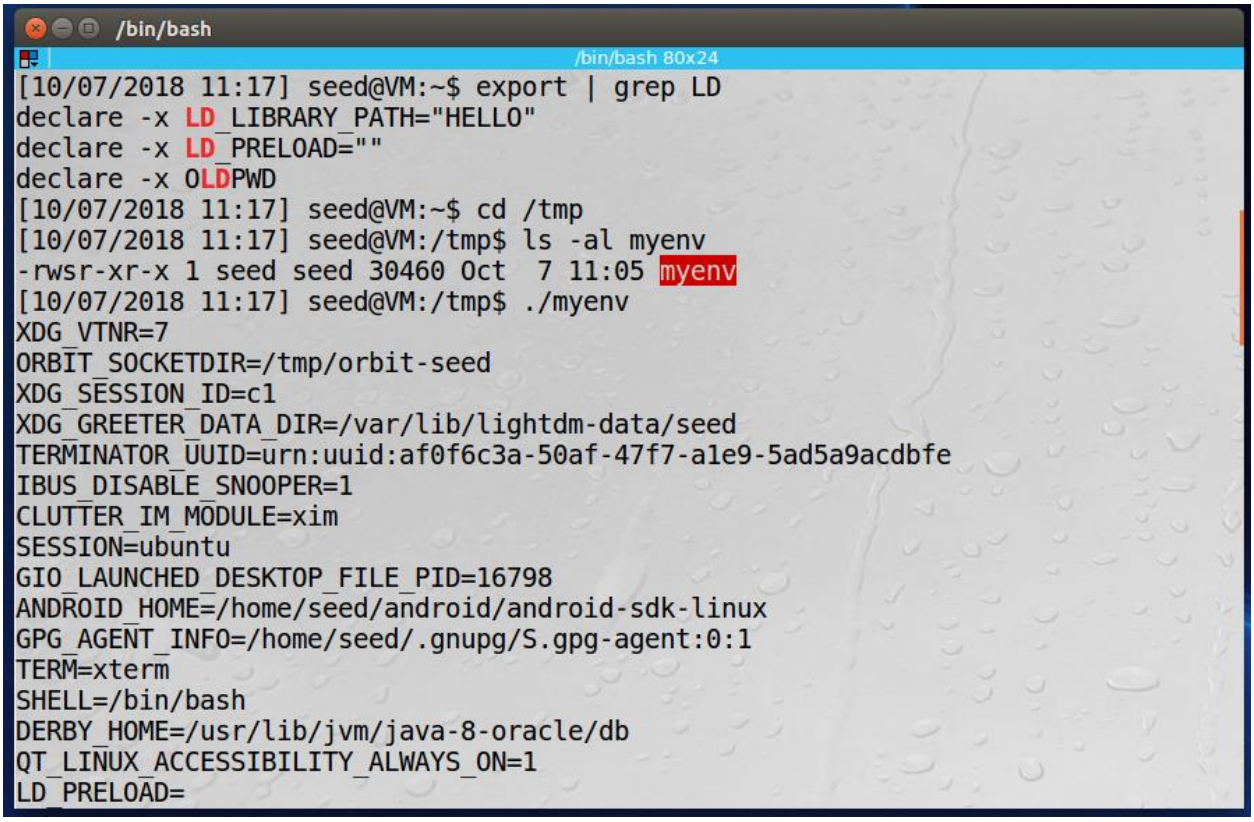
Scenario	System sleep function	DLL defined sleep function	Set-UID
#1 myprog, run as normal user		X	
#2 myprog Set-UID root, run as normal user	X		X
#3 myprog Set-UID root account		X	X
#4 myprog Set-UID as user1, executed as user2	X		X

It seems that in scenarios where we have Set-UID set and myprog is owned by someone else, the locally defined LD_PRELOAD does not get inherited / used by the executing process (myprog). This would lend itself to the conclusion that child processes do not inherit the LD_PRELOAD environment variable in cases where Set-UID and the owner and user are not the same. In the case where the current user and the

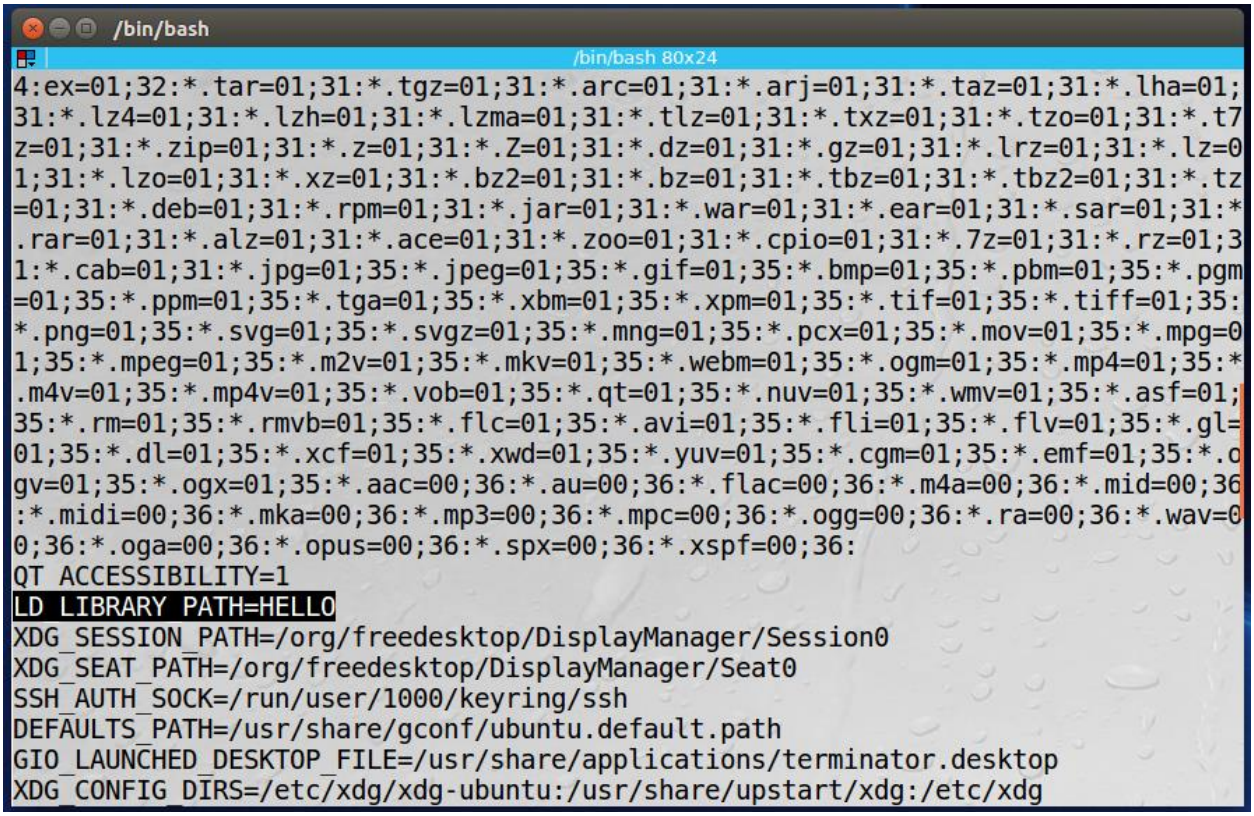
owner (of the program; e.g. myprog) are the same, the Set-UID seems to have no effect and the LD_PRELOAD is visible to the application.

Observations and Explanations (more investigation)

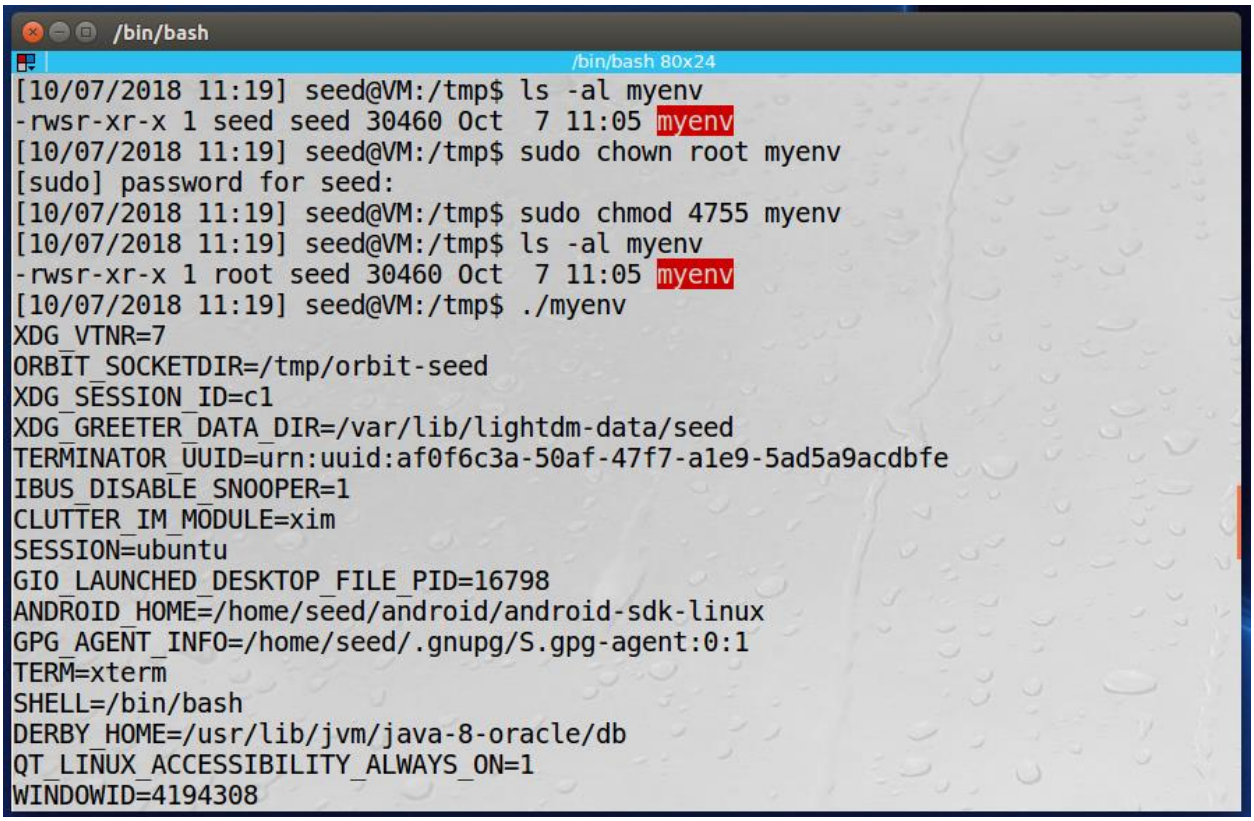
To investigate this further, I copied /usr/bin/env to /tmp/myenv. I set it to setuid. I was logged in as "seed". So, when running myenv, I can see the environment variable available to the program.



We can see in the screen capture below that even though we are a Set-UID program, we do indeed inherit the LD_LIBRARY_PATH variable (used LD_LIBRARY_PATH instead of LD_PRELOAD in this example). But how can this be? I believe since I am executing myenv as the user "seed" and the owner of myenv is also "seed", the Set-UID does not happen, therefore the environment that is in place for seed is fully inherited by the myenv application.



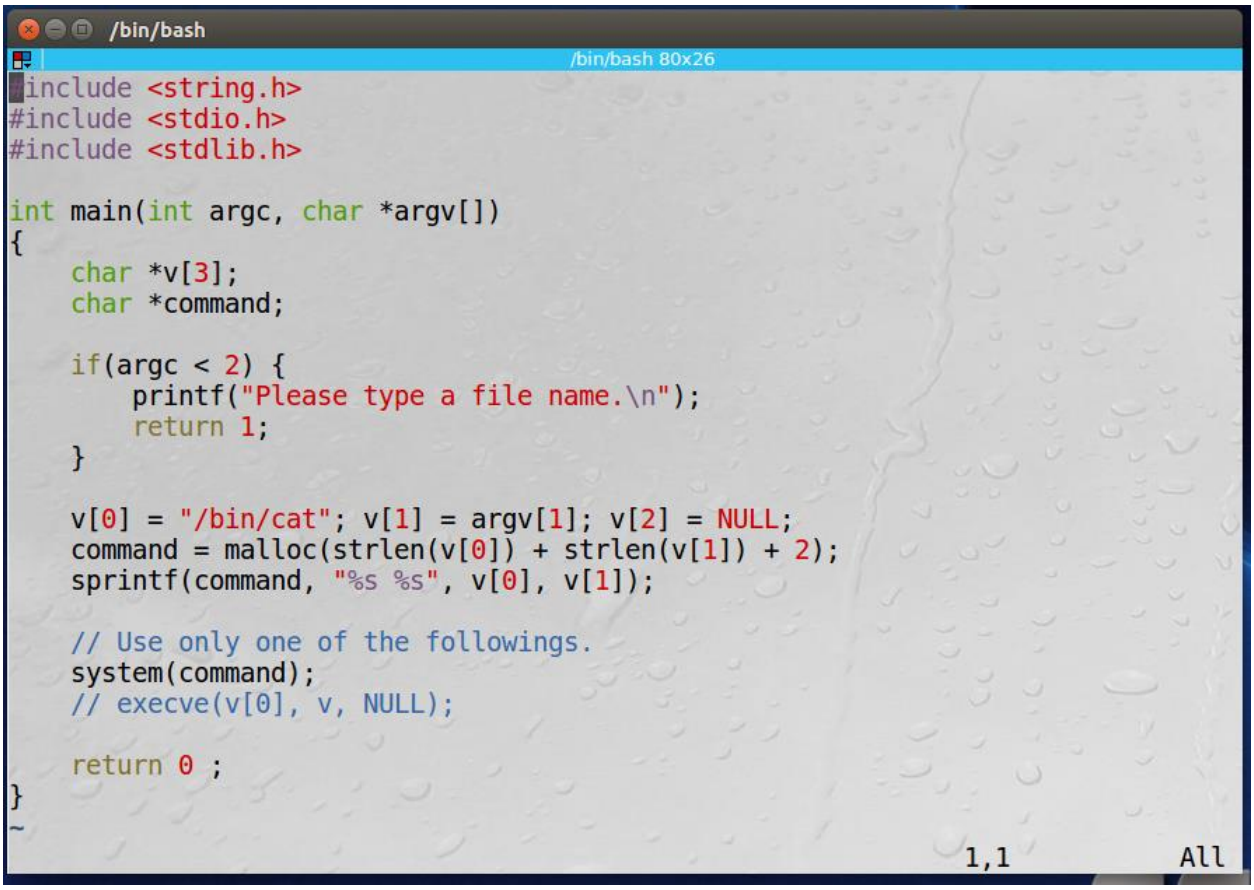
Doing the same experiment, but changing the owner to root, but still having it a Set-UID program, the LD_LIBRARY_PATH is not defined here.



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

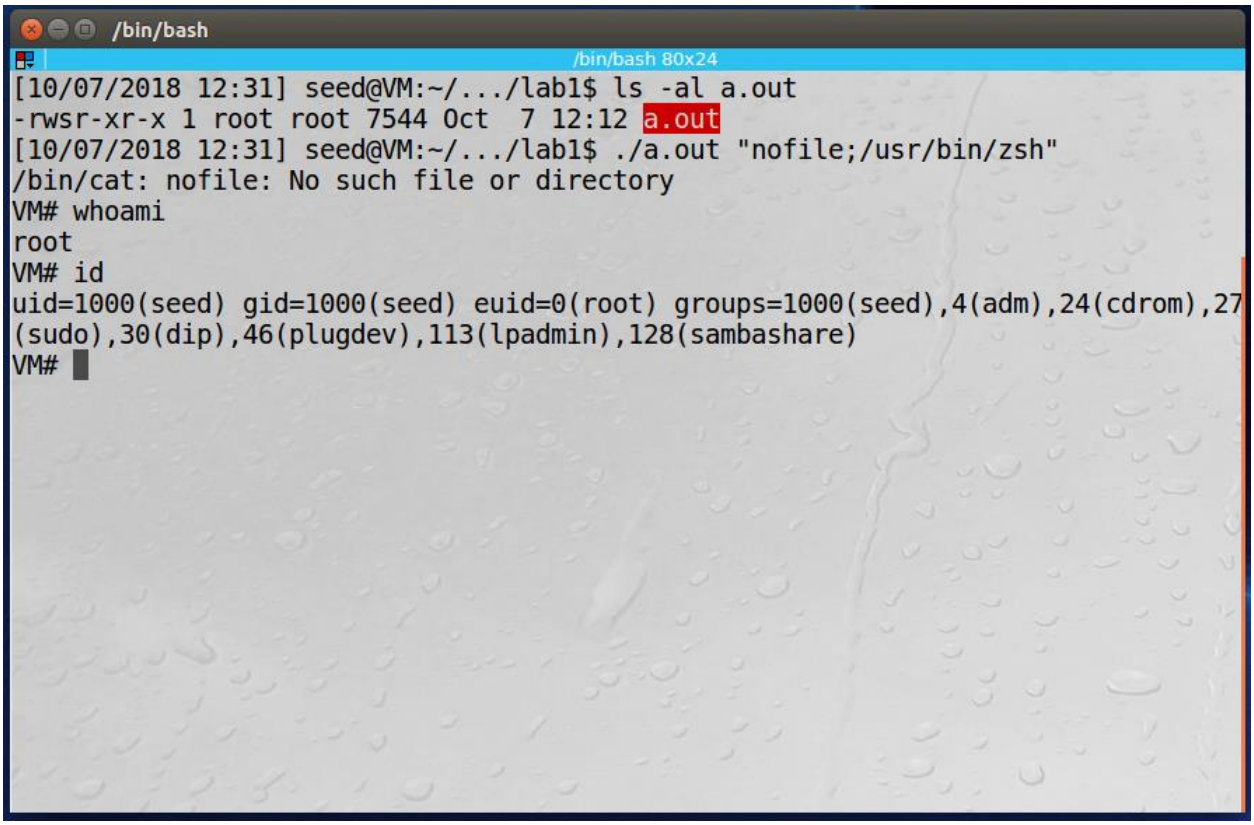
Execute the following program using system() and execve() and using an successful attack for system(), attempt it with execve().

Below is the program using system().

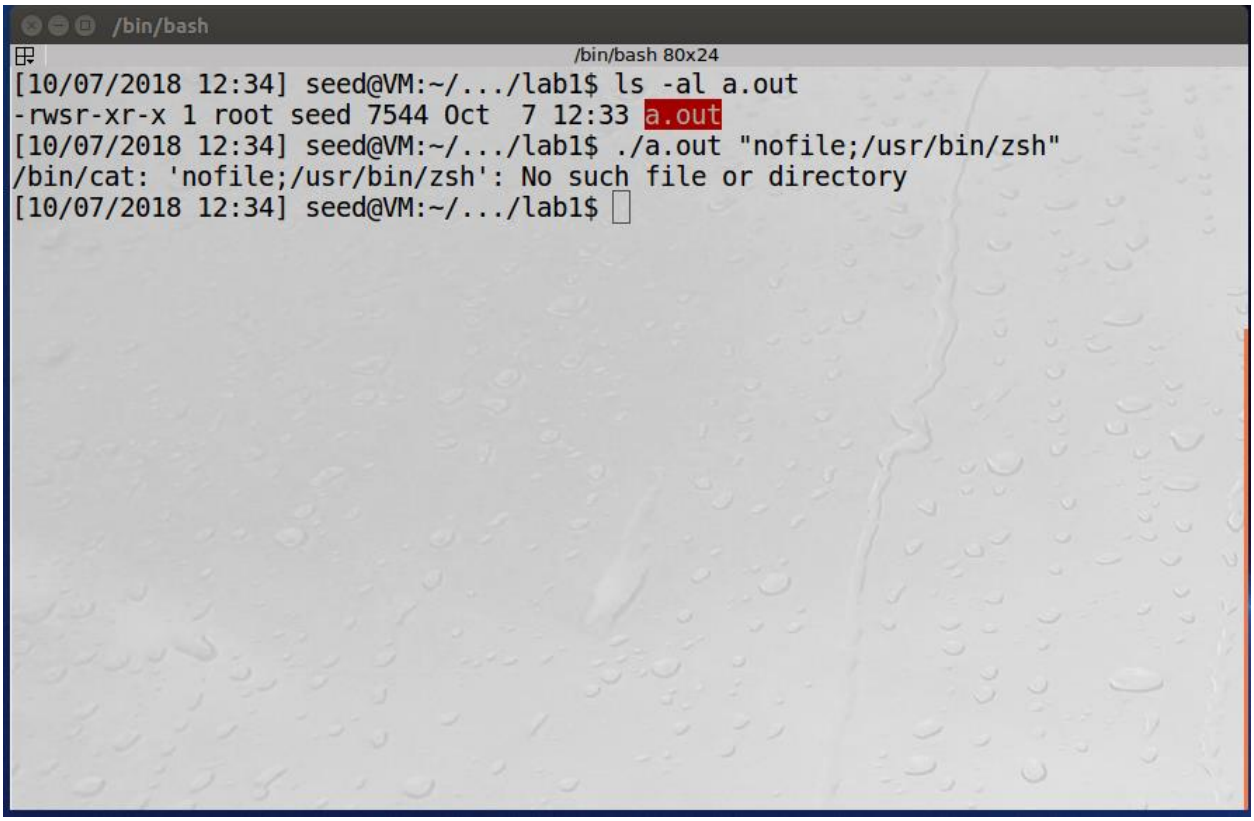


Observations and Explanations

For Step1, we build the program and set to Set-UID with root owner. We then attempt to execute /usr/bin/zsh. Please note, we still have /bin/sh symbolically linked to /bin/zsh as directed in Task 6. Anyway, when we execute the program with the command line argument "nofile;/usr/bin/zsh", we see that we execute a zsh shell with root permissions. Our effective id is root which allows us to do anything on the system, including removing any file we want, regardless of permission. The reason this attack works is that the system() executes a shell, so the parameter "nofile;/usr/bin/zsh" is passed to cat creating something like "cat nofile;/usr/bin/zsh". Since we are in a shell, "cat nofile" executes first and then "/usr/bin/zsh" executes which gives us root access.



In the screen shot below this, we try the same arguments using execve(). In this case, it does not attempt to launch a shell to execute the arguments. Instead it executes the cat program directly, not via a shell, so the argument is used as a literal to the command. In this case the command is 'cat' and the argument, which is the file name, is "nofile/usr/bin/zsh". Since cat cannot find this file, it ends with an error.

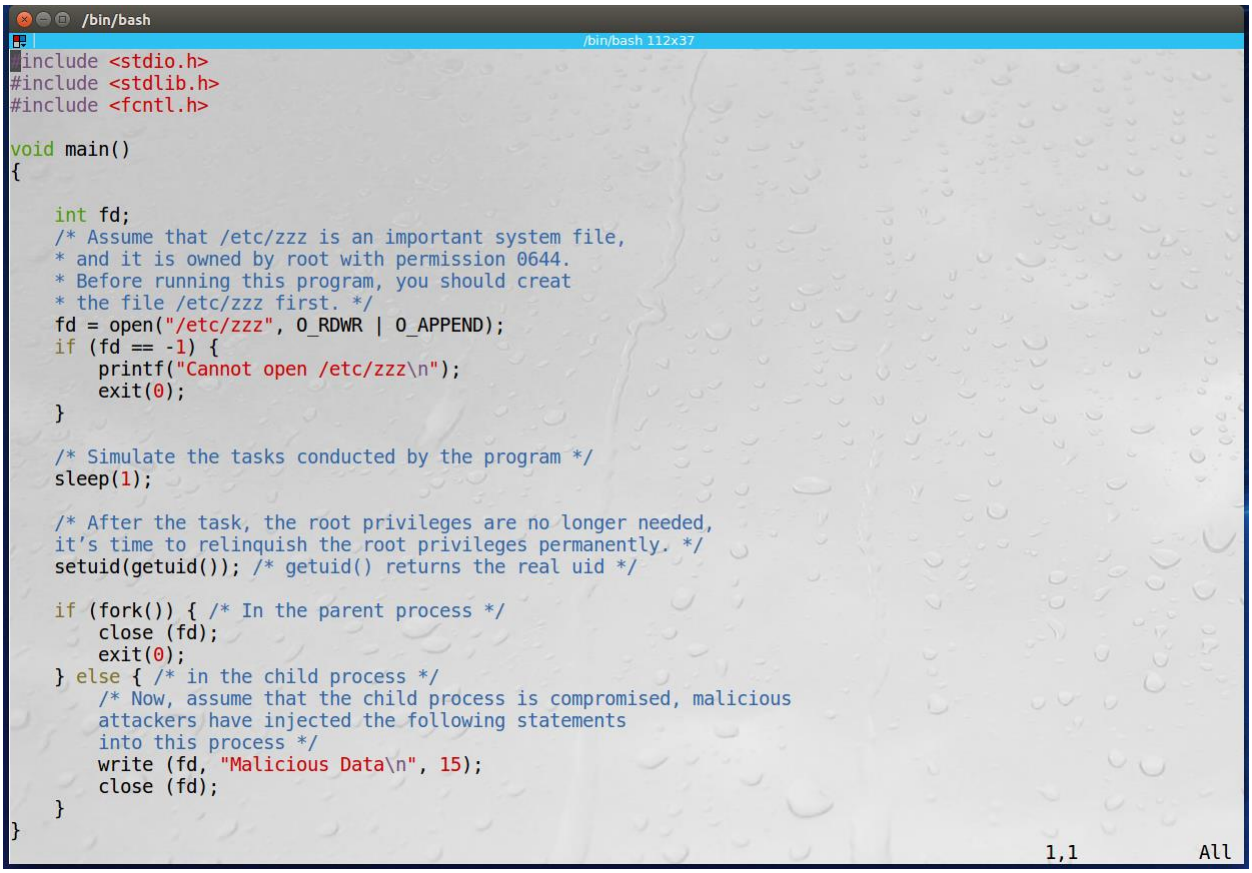


We can conclude that executing commands via "system()" can be very dangerous since appending any valid shell command to what's being passed in to, in this case, "cat" can allow for a system compromise. Using execve is much safer since it does not invoke a shell, so it is not vulnerable to these kinds of attacks.

Task 9: Capability Leaking

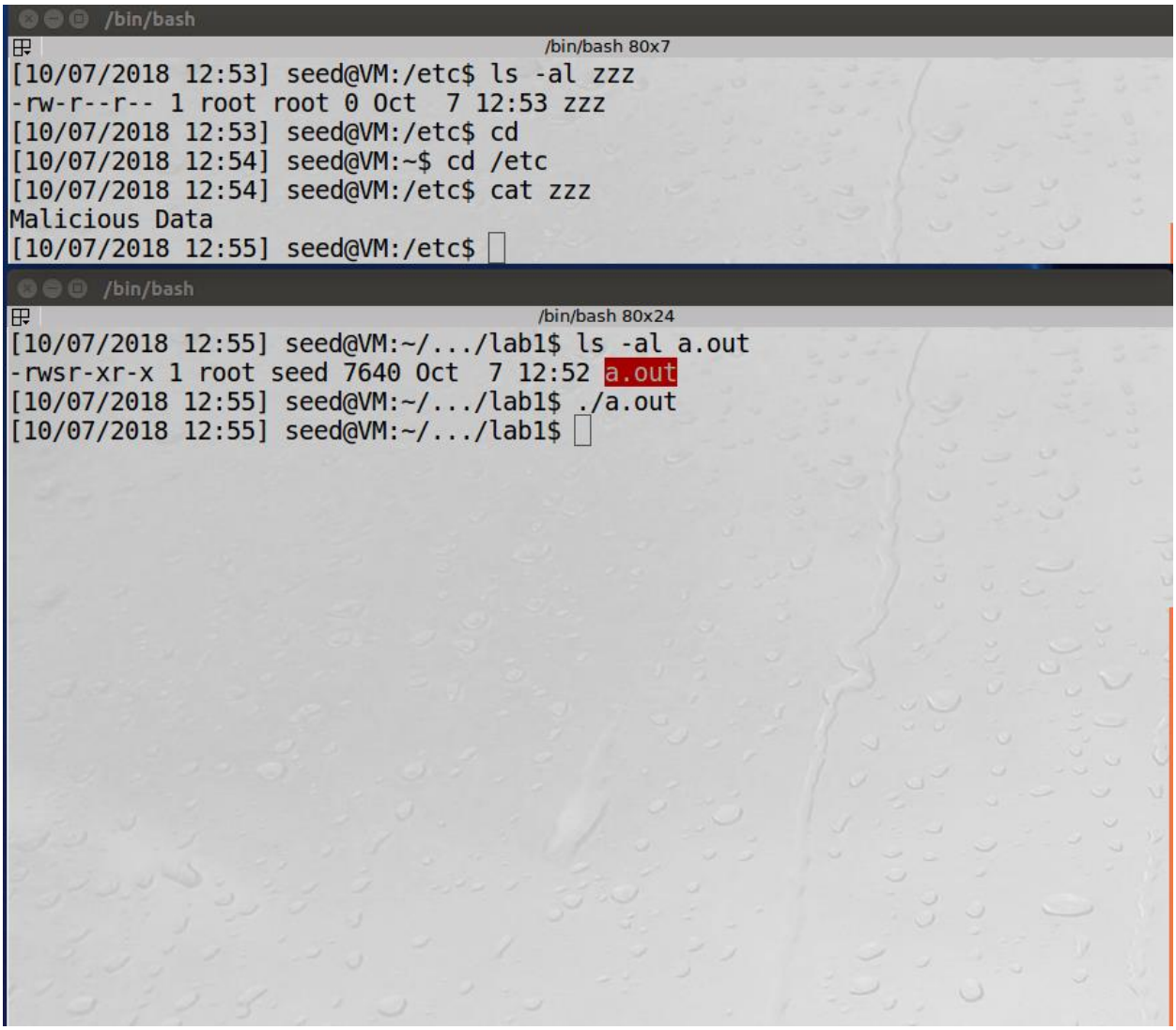
Compile the program below and observe the results to file /etc/zzz. Can the resource still be accessed even after the relinquishing Set-UID privalages?

Below is the program per Task 9. We previously created /etc/zzz which the program attempts to write to after relinquishing Set-UID.



Observations and Explanations

In the screen capture below, we have two parts. The bottom shows the permission for a.out and the top shows the /etc/zzz file contents. We can see that after executing of a.out, the zzz file contains the text "Malicious Data". How is this possible? This is possible because the program does indeed downgrade privileges by setting the uid to the real uid, but it does not close the file descriptor for the privileged file. Because of this, the file descriptor is available to the rest of the program. This is an interesting example since permissions are checked only once one the file is open, so if care is not taken to close this file or any privileged resources prior to relinquishing root privileges, those resource will still be available to the program. Which can be very dangerous!


/bin/bash
/bin/bash 80x7
[10/07/2018 12:53] seed@VM:/etc$ ls -al zzz
-rw-r--r-- 1 root root 0 Oct 7 12:53 zzz
[10/07/2018 12:53] seed@VM:/etc$ cd
[10/07/2018 12:54] seed@VM:~$ cd /etc
[10/07/2018 12:54] seed@VM:/etc$ cat zzz
Malicious Data
[10/07/2018 12:55] seed@VM:/etc$
/bin/bash
/bin/bash 80x24
[10/07/2018 12:55] seed@VM:~/.../lab1$ ls -al a.out
-rwsr-xr-x 1 root seed 7640 Oct 7 12:52 a.out
[10/07/2018 12:55] seed@VM:~/.../lab1$ ./a.out
[10/07/2018 12:55] seed@VM:~/.../lab1$