

CSCI 476: Lab 01

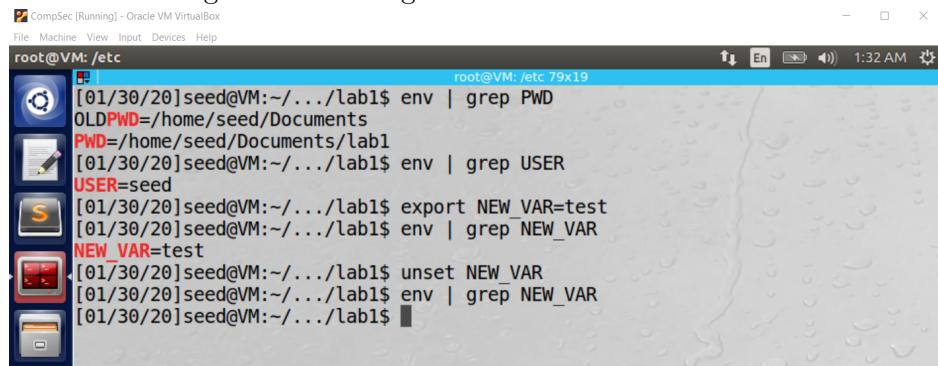
Nathan Stouffer

January 30, 2020

Task 1

In task 1, I familiarized myself with environment variables on linux command line. I did not provide a screenshot for all of my commands, but Figure 1 shows some experimentation.

Figure 1: Learning about environment variables



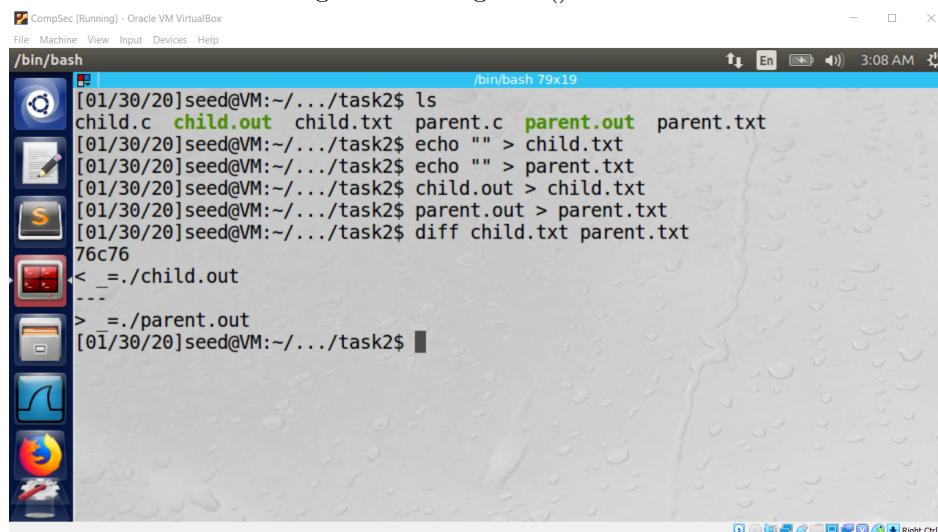
The screenshot shows a terminal window titled "CompSec [Running] - Oracle VM VirtualBox". The window title bar includes the application name, the host name "root@VM: /etc", the resolution "79x19", and the current time "1:32 AM". The terminal content displays the following sequence of commands and their outputs:

```
[01/30/20]seed@VM:~/.../lab1$ env | grep PWD
OLDPWD=/home/seed/Documents
PWD=/home/seed/Documents/lab1
[01/30/20]seed@VM:~/.../lab1$ env | grep USER
USER=seed
[01/30/20]seed@VM:~/.../lab1$ export NEW_VAR=test
[01/30/20]seed@VM:~/.../lab1$ env | grep NEW_VAR
NEW_VAR=test
[01/30/20]seed@VM:~/.../lab1$ unset NEW_VAR
[01/30/20]seed@VM:~/.../lab1$ env | grep NEW_VAR
[01/30/20]seed@VM:~/.../lab1$
```

Task 2

Task 2 focused on how a child process receives environment variables from the parent process. Below in Figure 2, there are two executable files with almost identical source code. The file parent.out prints out the environment variables of the parent process while child.out prints out the environment variables of a child process. The child process was created using the fork() method. In Figure 2, I first write an empty string to the two files child.txt and parent.txt, then I run each of the executable files (redirecting output to the appropriate file). Then, I used the diff command to see the differences in the file. The only difference is the name of the executable, so the environment variables must be the same. This means that, when using fork(), the child process inherits the environment variables of the parent process.

Figure 2: Using fork() method



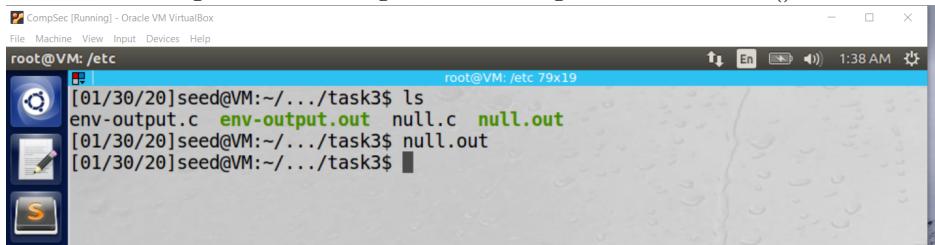
The screenshot shows a terminal window titled '/bin/bash' running on a virtual machine named 'CompSec [Running] - Oracle VM VirtualBox'. The terminal window has a title bar with icons for volume, brightness, and battery, and the time '3:08 AM'. The terminal content is as follows:

```
[01/30/20]seed@VM:~/.../task2$ ls  
child.c child.out child.txt parent.c parent.out parent.txt  
[01/30/20]seed@VM:~/.../task2$ echo "" > child.txt  
[01/30/20]seed@VM:~/.../task2$ echo "" > parent.txt  
[01/30/20]seed@VM:~/.../task2$ child.out > child.txt  
[01/30/20]seed@VM:~/.../task2$ parent.out > parent.txt  
[01/30/20]seed@VM:~/.../task2$ diff child.txt parent.txt  
76c76  
< ./child.out  
---  
> ./parent.out  
[01/30/20]seed@VM:~/.../task2$
```

Task 3

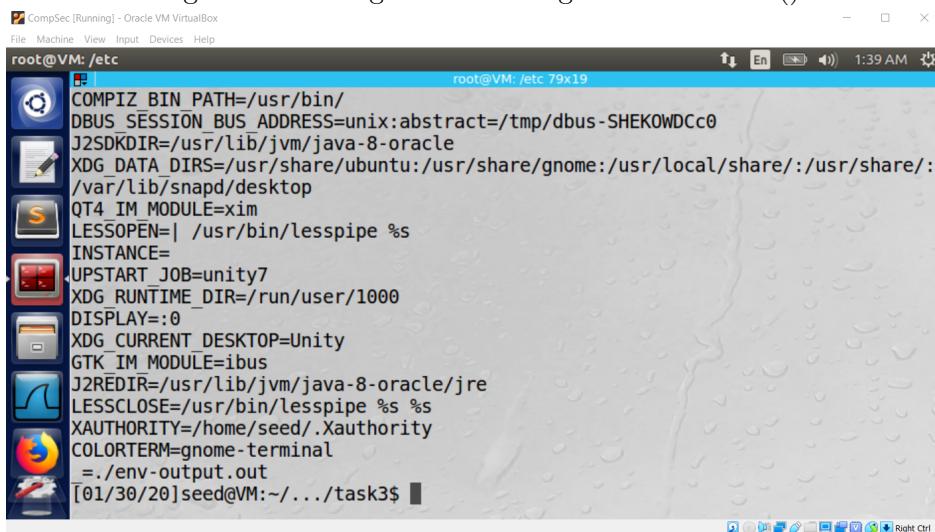
In task 3, I viewed how environment variables are affected when starting child processes via the execve() method. The executable null.out sends NULL as the final argument in execve() while env-output.out sends in the global variable environ. Each executable prints the environment variables of the child executable. As seen in Figure 3, null.out prints out nothing, so there are no environment variables. On the other hand, Figure 4 shows env-output.out to print the environment variables of the parent process. So, execve() must be told explicitly what its environment variables should be.

Figure 3: Sending NULL as argument in execve()



```
[root@VM: /etc] [01/30/20]seed@VM:~/.task3$ ls
env-output.c env-output.out null.c null.out
[01/30/20]seed@VM:~/.task3$ null.out
[01/30/20]seed@VM:~/.task3$
```

Figure 4: Sending environ as argument in execve()

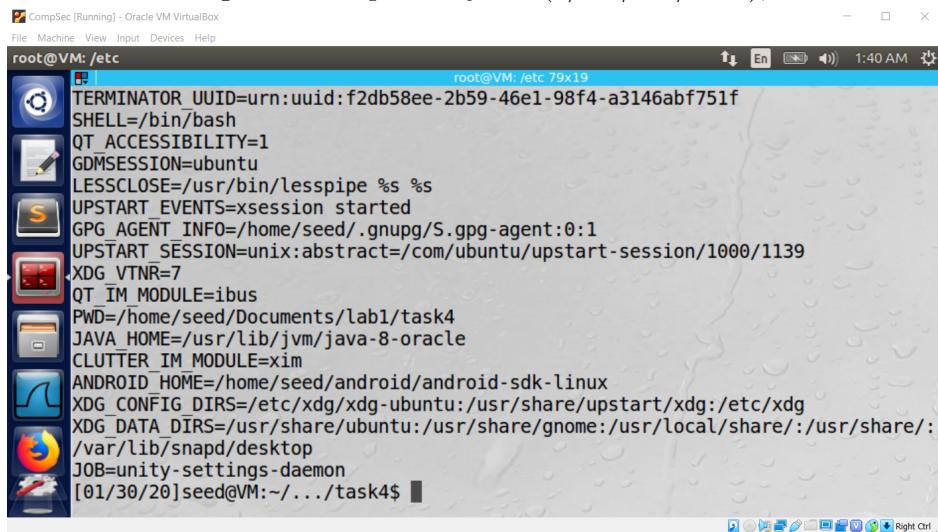


```
[root@VM: /etc] [01/30/20]seed@VM:~/.task3$ ls
COMPIZ_BIN_PATH=/usr/bin/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-SHEKOWDCc0
J2SDKDIR=/usr/lib/jvm/java-8-oracle
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:
/var/lib/snapd/desktop
QT4_IM_MODULE=xim
LESSOPEN=| /usr/bin/lesspipe %s
INSTANCE=
UPSTART_JOB=unity7
XDG_RUNTIME_DIR=/run/user/1000
DISPLAY=:0
XDG_CURRENT_DESKTOP=Unity
GTK_IM_MODULE=ibus
J2REDIR=/usr/lib/jvm/java-8-oracle/jre
LESSCLOSE=/usr/bin/lesspipe %s %s
XAUTHORITY=/home/seed/.Xauthority
COLORTERM=gnome-terminal
= ./env-output.out
[01/30/20]seed@VM:~/.task3$
```

Task 4

Task 4 is a simple test of using `system()` to execute other programs. The output of the call `system("/usr/bin/env")` is shown in Figure 5. This verifies that `system()` starts a child process since a new process (a shell) is created and the environment variables of the parent process are printed. So the calling process must be a child.

Figure 5: Output of `system("/usr/bin/env")`;

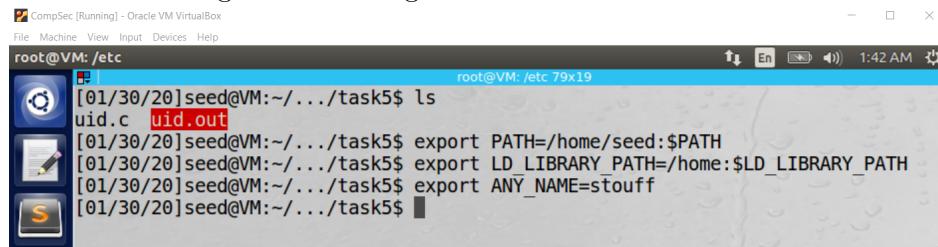


```
root@VM: /etc
TERMINATOR UUID=urn:uuid:f2db58ee-2b59-46e1-98f4-a3146abf751f
SHELL=/bin/bash
QT_ACCESSIBILITY=1
GDMSESSION=ubuntu
LESSCLOSE=/usr/bin/lesspipe %s %s
UPSTART_EVENTS=xsession started
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1139
XDG_VTNR=7
QT_IM_MODULE=ibus
PWD=/home/seed/Documents/lab1/task4
JAVA_HOME=/usr/lib/jvm/java-8-oracle
CLUTTER_IM_MODULE=xim
ANDROID_HOME=/home/seed/android/android-sdk-linux
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:
/var/lib/snapd/desktop
JOB=unity-settings-daemon
[01/30/20]seed@VM:~/.../task4$
```

Task 5

Task 5 investigates how Set-UID programs are affected by environment variables. In Figure 6, I messed with some environment variables in the shell. I then ran the Set-UID program uid.out. The executable uid.out runs with root privilege, but is vulnerable. This is because Figures 7 and 8 show that the environment variables in the parent process get passed into the child process. For some reason, I could not find LD_LIBRARY_PATH in the output, but PATH and ANY_NAME most definitely appeared. The LD* issue is discussed further in Task 7.

Figure 6: Messing with environment variables



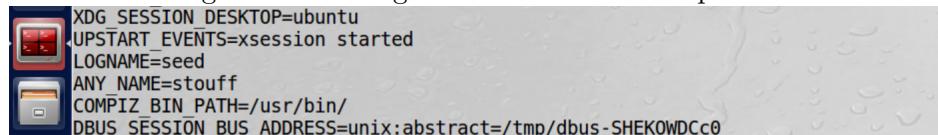
```
CompSec [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@VM: /etc
[01/30/20]seed@VM:~/.../task5$ ls
uid.c uid.out
[01/30/20]seed@VM:~/.../task5$ export PATH=/home/seed:$PATH
[01/30/20]seed@VM:~/.../task5$ export LD_LIBRARY_PATH=/home:$LD_LIBRARY_PATH
[01/30/20]seed@VM:~/.../task5$ export ANY_NAME=stouff
[01/30/20]seed@VM:~/.../task5$
```

Figure 7: Showing PATH in child process



```
GIO_LAUNCHED_DESKTOP_FILE=/usr/share/applications/terminator.desktop
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
DESKTOP_SESSION=ubuntu
PATH=/home/seed:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/bin:/sbin:/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/bin:/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
QT_IM_MODULE=ibus
```

Figure 8: Showing ANY_NAME in child process

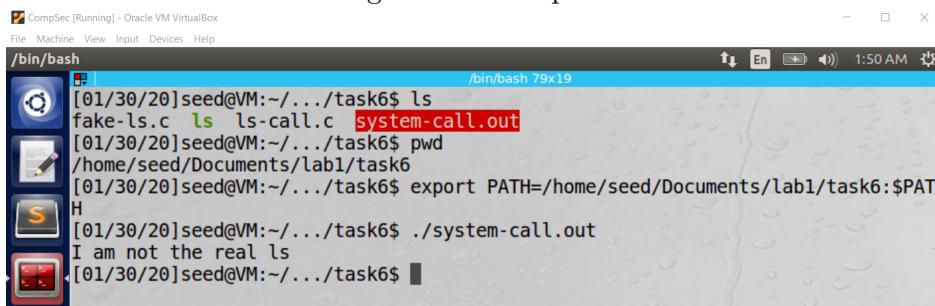


```
XDG_SESSION_DESKTOP=ubuntu
UPSTART_EVENTS=xsession started
LOGNAME=seed
ANY_NAME=stouff
COMPIZ_BIN_PATH=/usr/bin/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-SHEKOWDCc0
```

Task 6

Task 6 is an example of how changing the environment variables can affect the behavior of a program (even a Set-UID program). The executable system-call.out uses the command system("ls") to run the ls command. However, the absolute path of the command is not specified. So, I created a new executable called ls with different functionality. Then, I changed the PATH environment variable so that programs searched the location of my ls executable before looking elsewhere. Figure 9 shows that system-call.out used the functionality of my false ls command since it did not specify the absolute file path for ls.

Figure 9: test caption



```
CompSec [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
/bin/bash 79x19
[01/30/20]seed@VM:~/.../task6$ ls
fake-ls.c ls ls-call.c system-call.out
[01/30/20]seed@VM:~/.../task6$ pwd
/home/seed/Documents/lab1/task6
[01/30/20]seed@VM:~/.../task6$ export PATH=/home/seed/Documents/lab1/task6:$PATH
[01/30/20]seed@VM:~/.../task6$ ./system-call.out
I am not the real ls
[01/30/20]seed@VM:~/.../task6$
```

Task 7

Task 7 looks at another way changing environment variables can affect the way programs run. In Figure 10, I dynamically link a library that overwrites the sleep(int s) function. I then add that library to the LD_PRELOAD environment variable so that any executables will look at the ./libmylib.so.1.0.1 before searching other libraries. This means if sleep(int s) is called, my code will run. Figure 10 shows this to be true for a regular program. However, it also shows that this is not the case for a root-owned Set-UID program. The executable myuidprog.out runs sleep(int s) as the expected sleep function.

Figure 10: Compiling library and running executables

The screenshot shows a terminal window titled 'CompSec [Running] - Oracle VM VirtualBox'. The terminal is running a bash shell. The user has compiled a C program named 'myprog.c' into an executable 'myprog.out'. They then created a shared library 'mylib.o' and linked it into 'libmylib.so.1.0.1'. They export the LD_PRELOAD environment variable to point to this library. When they run 'myprog.out', it prints 'I am not sleeping!' instead of sleeping. Finally, they run 'myuidprog.out', which is a root-owned program, and it correctly sleeps for 1 second as expected.

```
[01/30/20]seed@VM:~/.../task7$ ls  
mylib.c myprog.c myprog.out myuidprog.out  
[01/30/20]seed@VM:~/.../task7$ gcc -fPIC -g -c mylib.c  
[01/30/20]seed@VM:~/.../task7$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc  
[01/30/20]seed@VM:~/.../task7$ export LD_PRELOAD=./libmylib.so.1.0.1  
[01/30/20]seed@VM:~/.../task7$ env | grep LD_PRELOAD  
LD_PRELOAD=./libmylib.so.1.0.1  
[01/30/20]seed@VM:~/.../task7$ ./myprog.out  
I am not sleeping!  
[01/30/20]seed@VM:~/.../task7$ ./myuidprog.out  
[01/30/20]seed@VM:~/.../task7$
```

In Figure 11, I change to the root user where I again export the LD_PRELOAD variable. Now running the root-owned file myuidprog.out runs my false code instead of the standard sleep(int s).

Figure 11: Running executable as root owner/user

The screenshot shows a terminal window titled 'CompSec [Running] - Oracle VM VirtualBox'. The terminal is running as the root user. The user exports the LD_PRELOAD environment variable to point to the dynamically linked library 'libmylib.so.1.0.1'. When they run 'myuidprog.out', it prints 'I am not sleeping!' instead of sleeping, demonstrating that the child process receives the parent's environment variables.

```
root@VM: /home/seed/Documents/lab1/task7$ ls  
libmylib.so.1.0.1 mylib.c mylib.o myprog.c myprog.out myuidprog.out  
root@VM: /home/seed/Documents/lab1/task7$ su root  
Password:  
root@VM: /home/seed/Documents/lab1/task7# export LD_PRELOAD=./libmylib.so.1.0.1  
root@VM: /home/seed/Documents/lab1/task7# env | grep LD_PRELOAD  
LD_PRELOAD=./libmylib.so.1.0.1  
root@VM: /home/seed/Documents/lab1/task7# ./myuidprog.out  
I am not sleeping!  
root@VM: /home/seed/Documents/lab1/task7#
```

In Figure 12, I change to a temporary user called tmp. I again export the LD_PRELOAD variable. Now running the tmp-owned file myuidprog.out (note that this is a Set-UID program) runs my code in the dynamically linked library instead of the standard sleep(int s).

Based on the examples in this test. It appears that the function sleep(int s) in my dynamically linked library is called when the user executing is the same as the owner. As evidence for this, the only situation where "I am not sleeping!" was not printed to the console was when seed ran root-owned program myuidprog.out. Based on this, it seems that, for Set-UID programs, the child process does not receive LD* environment variables from the parent, but instead from the owner's environment. To test this, we will use the code from Task 4 and reference Task 5.

In Figure 13, I change the ownership of the executable system-test.out to root and export a LD_PRELOAD variable. As seen in Figure 14, the variable LD_PRELOAD is printed out.

Figure 12: Running executable with tmp as owner

```
[01/30/20]seed@VM:~/.../task7$ su tmp
root@VM: /home/seed/Documents/lab1/task7
[01/30/20]root@VM:~/.../task7$ ls -l
total 36
-rwxrwxr-x 1 seed seed 7936 Jan 30 10:25 libmylib.so.1.0.1
-rw-rw-r-- 1 seed seed 223 Jan 29 11:22 mylib.c
-rw-rw-r-- 1 seed seed 2596 Jan 30 10:25 mylib.o
-rw-rw-r-- 1 seed seed 63 Jan 30 01:20 myprog.c
-rwxrwxr-x 1 seed seed 7348 Jan 30 10:20 myprog.out
-rwsr-xr-x 1 tmp seed 7348 Jan 30 10:10 myuidprog.out
tmp@VM:/home/seed/Documents/lab1/task7$ export LD_PRELOAD=./libmylib.so.1.0.1
tmp@VM:/home/seed/Documents/lab1/task7$ ./myuidprog.out
I am not sleeping!
tmp@VM:/home/seed/Documents/lab1/task7$
```

However, recall that system-call.out is not a Set-UID program, it is only owned by root. In Task 5, I noted that I could not find the LD_LIBRARY_PATH variable in the environment variables output. Additionally, the executable ran in Task 5 was a Set-UID program. So this fits with my hypothesis that Set-UID programs ignore LD* variables of their parent and inherit from their owner. If this were not the case, then an attacker could easily manipulate a root-owned program with a false dynamically linked library and abuse root power. So, if I am correct, then it is a good security measure.

Figure 13: Showing commands

```
[01/30/20]seed@VM:~/.../task4$ sudo chown root system-test.out
[01/30/20]seed@VM:~/.../task4$ ls -l
total 12
-rw-rw-r-- 1 seed seed 184 Jan 29 01:27 system-test.c
-rwxrwxr-x 1 root seed 7352 Jan 30 01:06 system-test.out
[01/30/20]seed@VM:~/.../task4$ export LD_PRELOAD=testlib
ERROR: ld.so: object 'testlib' from LD_PRELOAD cannot be preloaded (cannot open
shared object file): ignored.
[01/30/20]seed@VM:~/.../task4$
```

Figure 14: test

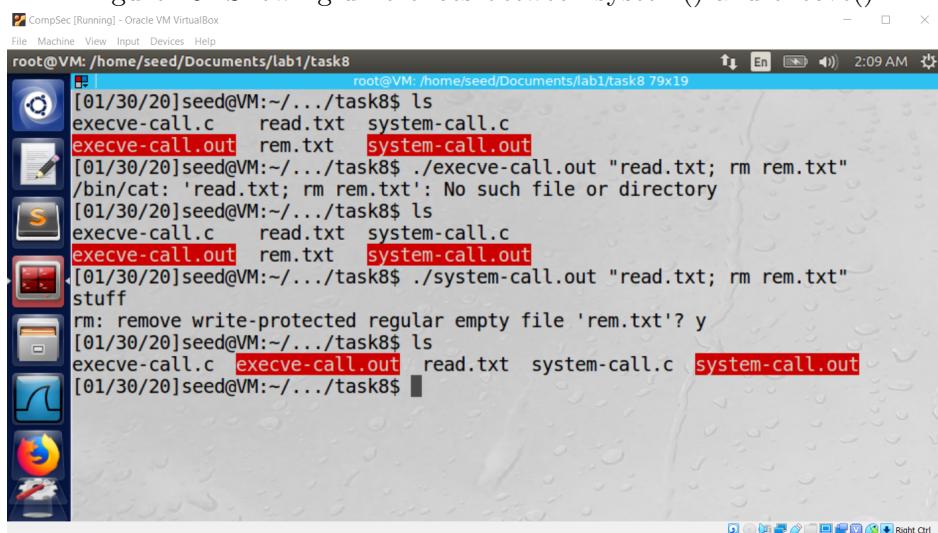
```
COMPIZ_BIN_PATH=/usr/bin/
DISPLAY=:0
IBUS_DISABLE_SNOOPER=1
LD_PRELOAD=testlib
LANG=en_US.UTF-8
XDG_CURRENT_DESKTOP=Unity
```

This is an especially dangerous attack since the attacker does not even have to really know what source code entails. As an attacker, you could dynamically link a common function like printf() and then overwrite that function to do whatever you wanted (so long as you are able to add the library to the LD_PRELOAD environment variable).

Task 8

Task 8 shows a vulnerability in calling `system()` that does not exist when calling `execve()`. In Figure 15, each executable has the same intention. A user runs the program with a file name as an argument, the executable then prints the contents of that file to the screen. However, one executable uses `system()` while the other uses `execve()`. For `execve()`, if a user were to try to tack on extra commands to the end of their file name, they would get an error (see Figure 15). While in `system()`, a user is free to tack on extra commands. This is because `system()` runs a full on command line, so the extra commands are executed as a terminal would execute them. This is made even more dangerous by the fact the `system-call.out` is a Set-UID program and is able to have full access any file (since they would have root privileges). I was even able to remove the write protected file `rem.txt`. In the context of the auditing example, the auditor would be able to do whatever they wanted to the system. On the other hand, `execve()` completely separates its commands and arguments.

Figure 15: Showing differences between `system()` and `execve()`

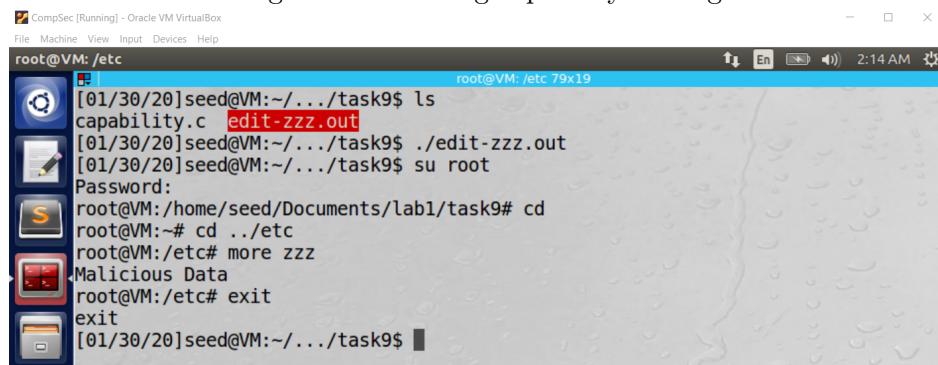


```
root@VM: /home/seed/Documents/lab1/task8
[01/30/20]seed@VM:~/.../task8$ ls
execve-call.c  read.txt  system-call.c
execve-call.out  rem.txt  system-call.out
[01/30/20]seed@VM:~/.../task8$ ./execve-call.out "read.txt; rm rem.txt"
/bin/cat: 'read.txt; rm rem.txt': No such file or directory
[01/30/20]seed@VM:~/.../task8$ ls
execve-call.c  read.txt  system-call.c
execve-call.out  rem.txt  system-call.out
[01/30/20]seed@VM:~/.../task8$ ./system-call.out "read.txt; rm rem.txt"
stuff
rm: remove write-protected regular empty file 'rem.txt'? y
[01/30/20]seed@VM:~/.../task8$ ls
execve-call.c  execve-call.out  read.txt  system-call.c  system-call.out
[01/30/20]seed@VM:~/.../task8$
```

Task 9

Task 9 focuses on capability leaking. A program can change its privileges at run-time by using the setuid() function. In edit-zzz.out, setuid() is used to change the effective user id to the real user id (that of the owner). While it does this successfully, setuid() does not tie up all the loose ends. For example, in edit-zzz.out opens etc/zzz with root permissions and does not close it before downgrading privileges. This means that, even with root privileges revoked, edit-zzz.out can continue to change the root file zzz. This is shown below in Figure 16.

Figure 16: Showing capability leaking



The screenshot shows a terminal window titled 'CompSec [Running] - Oracle VM VirtualBox'. The terminal session is as follows:

```
[01/30/20]seed@VM:~/.../task9$ ls
capability.c edit-zzz.out
[01/30/20]seed@VM:~/.../task9$ ./edit-zzz.out
[01/30/20]seed@VM:~/.../task9$ su root
Password:
root@VM:/home/seed/Documents/lab1/task9# cd ..
root@VM:~# cd /etc
root@VM:/etc# more zzz
Malicious Data
root@VM:/etc# exit
exit
[01/30/20]seed@VM:~/.../task9$
```

The terminal shows a user named 'seed' running a script named 'edit-zzz.out' which appears to be a exploit for task9. It then switches to root privileges and writes 'Malicious Data' to the file '/etc/zzz'. Finally, it exits the root shell.