Ethan Booker

Environment Variable and Set-UID Program Lab

2/21/2020

Description of the overall goals of the lab

The overall goal of this lab is to gain further insight on environment variables, particularly about how they work, propagate, affect system/program behaviors (specifically behavior of Set-UID programs), and how they can lead to vulnerabilities if not handled properly.

Section 2.1

Task 1: Manipulating Environment Variables

Description: Learn to set and unset program variables.

Evidence: No screenshot. Only commands. The reason why is because other steps will require I use these commands, so either way you know that I can do these.

- Called: unset LANGUAGE
    - Removed it from the environment variable list
- Called: export LANGUAGE=en_US
    - Add this variable back into the environment variable list

Analysis: unset LANGUAGE, looked for the LANGUAGE variable and removed it from the list. When I called printenv, LANGUAGE was no longer there. I then called export LANGUAGE=en_US and the LANGUAGE variable was back in the environment list. The reason for these behaviors is because unset and export are special bash commands that allow us to manipulate the environment list from the command line.

Section 2.2

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

Description: Create a process using fork() to see how the child process gets its environment variables.

Evidence:

```
/bin/bash 57x23
[02/21/20]seed@VM:~$ ls
2.2.1.out    c.out              myprog2.c
2.2.1.txt    Customization      myprog2.out
2.2.c        d.c                myprog3.c
2.2.out      Desktop            myprog3.out
a2.out       Documents          myprog.out
a.c          d.out              passwd_input
android      Downloads          Pictures
a.out        e.c                Public
attack.sh    e.out              source
b.c          examples.desktop   Templates
bChild.txt   lib                Videos
bin          libmylib.so.1.0.1  vulp
b.out        Music              vulp.c
c.c          mylib.c
child.txt    mylib.o
[02/21/20]seed@VM:~$ 2.2.out > 2.2.txt
[02/21/20]seed@VM:~$ diff 2.2.txt 2.2.1.txt
77c77
< _=./2.2.out
---
> _=./2.2.1.out
[02/21/20]seed@VM:~$ 
```

Analysis: There is almost no difference in the files besides this hex number: 77c77 which could represent some type of ID. But I got similar files because the child process inherited the environment variables from the parent process then they have the same environment variables.

Section 2.3

Task 3: Environment Variables and execve()

Description: Learned how environment variables are affected when execve runs.

Evidence: No screenshot. Once again, other programs use these functions, I thought this was a bit redundant and thus self-explanatory based on my analysis.

- When I ran the program with **execve("/usr/bin/env", argv, NULL);**
  - **nothing showed up.**
- When I changed the program to **execve("/usr/bin/env", argv, environ);**
  - **A list of my environment variables showed up.**

Analysis: Basically what happened was that the **execve** function didn't print anything out due to it's arguments being null. But, when we changed the arguments to a valid char array then the contents of that char which happened to be the environments list to be printed out.

Section 2.4

Task 4: Environment Variables and system()

Description: See how environment variables are affected by the system() call.

Evidence:

```
/bin/bash 57x23
  }
}
[02/21/20]seed@VM:~$ cat e.c
int main()
{
  system("ls");
  return 0;
}
[02/21/20]seed@VM:~$ cat c.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  system("/usr/bin/env");
  return 0;
}
[02/21/20]seed@VM:~$ ./c.out
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
IBUS_DISABLE_SNOOPER=1
```

Analysis: system() calls a command for the shell to execute the command, which is **execl()** and that function calls **execve**() while passing in the environment variables array. Which is why a list of environment variables appear.

Section 2.5

Task 5: Environment Variable and Set-UID Programs

Description: Learn to use Set-UID and how Set-UID programs are affected.

Evidence:

```
user1@VM: /home/seed 87x23

extern char **environ;

void main()
{
  int i = 0;
  while (environ[i] != NULL) {
    printf("%s\n", environ[i]);
    i++;
  }
}
[02/21/20]seed@VM:~$ su user
No passwd entry for user 'user'
[02/21/20]seed@VM:~$ su user1
Password:
user1@VM:/home/seed$ export MY_ENV="my env"
user1@VM:/home/seed$ export PATH="my env":$PATH
user1@VM:/home/seed$ ./d.out | grep "my env"
PATH=my env:/home/seed/bin:/usr/local/sbin:/usr/local/bin
:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/game
s:.
MY_ENV=my env
user1@VM:/home/seed$ 
```

```
root@VM: /home/seed 69x38
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_CONFIG_PROFILE=ubuntu-lowgfx
IM_CONFIG_PHASE=1
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
GTK2_MODULES=overlay-scrollbar
SHLVL=1
HOME=/home/seed
XDG_SEAT=seat0
LANGUAGE=en_US
LIBGL_ALWAYS_SOFTWARE=1
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
UPSTART_INSTANCE=
XDG_SESSION_DESKTOP=ubuntu
UPSTART_EVENTS=xsession started
LOGNAME=seed
ANY_NAME=nope
COMPIZ_BIN_PATH=/usr/bin/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-oUDZjf6xml
J2SDKDIR=/usr/lib/jvm/java-8-oracle
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/u
sr/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
_=./d.out
[02/21/20]seed@VM:~$ 
```

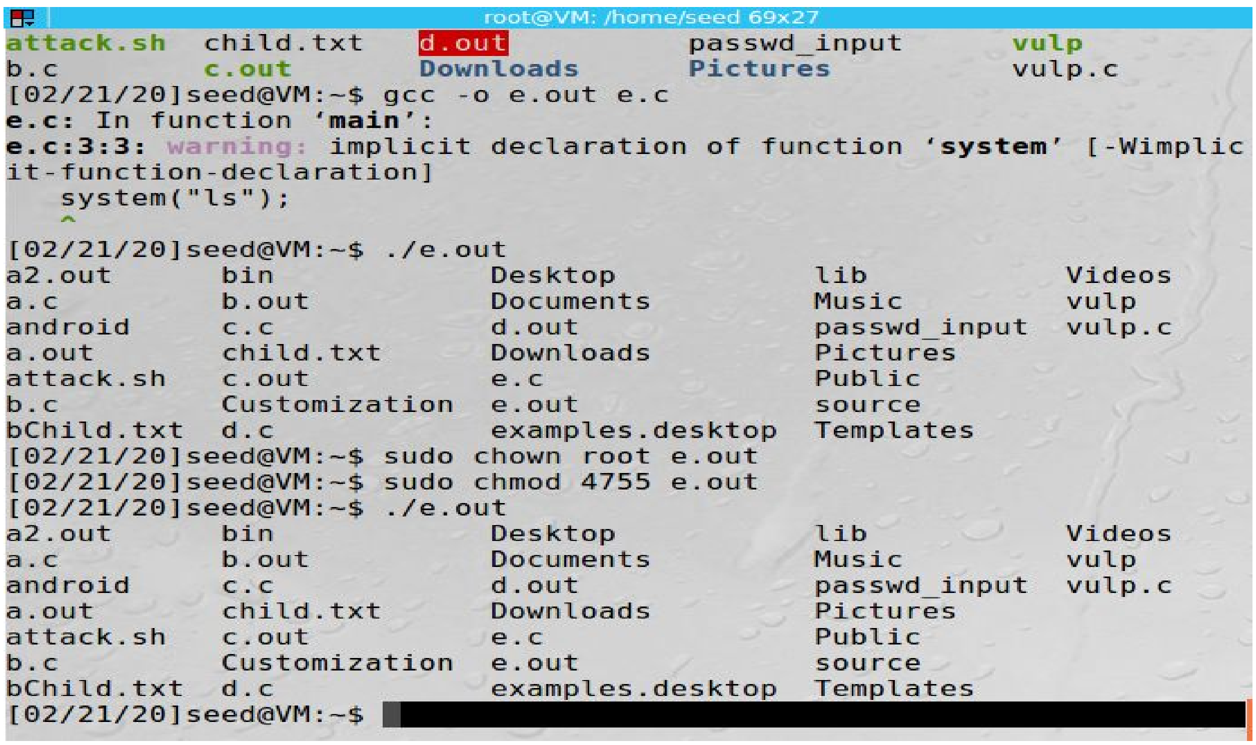
Analysis: Since the program's Set-UID was on root, as a normal user, I could use that program and look at all my environment variables along with the new ones that I set. This happened because of Set-UID giving the program root privileges during execution.

Section 2.6

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

Description: Learn to use Set-UID and how Set-UID programs are affected.

Evidence:



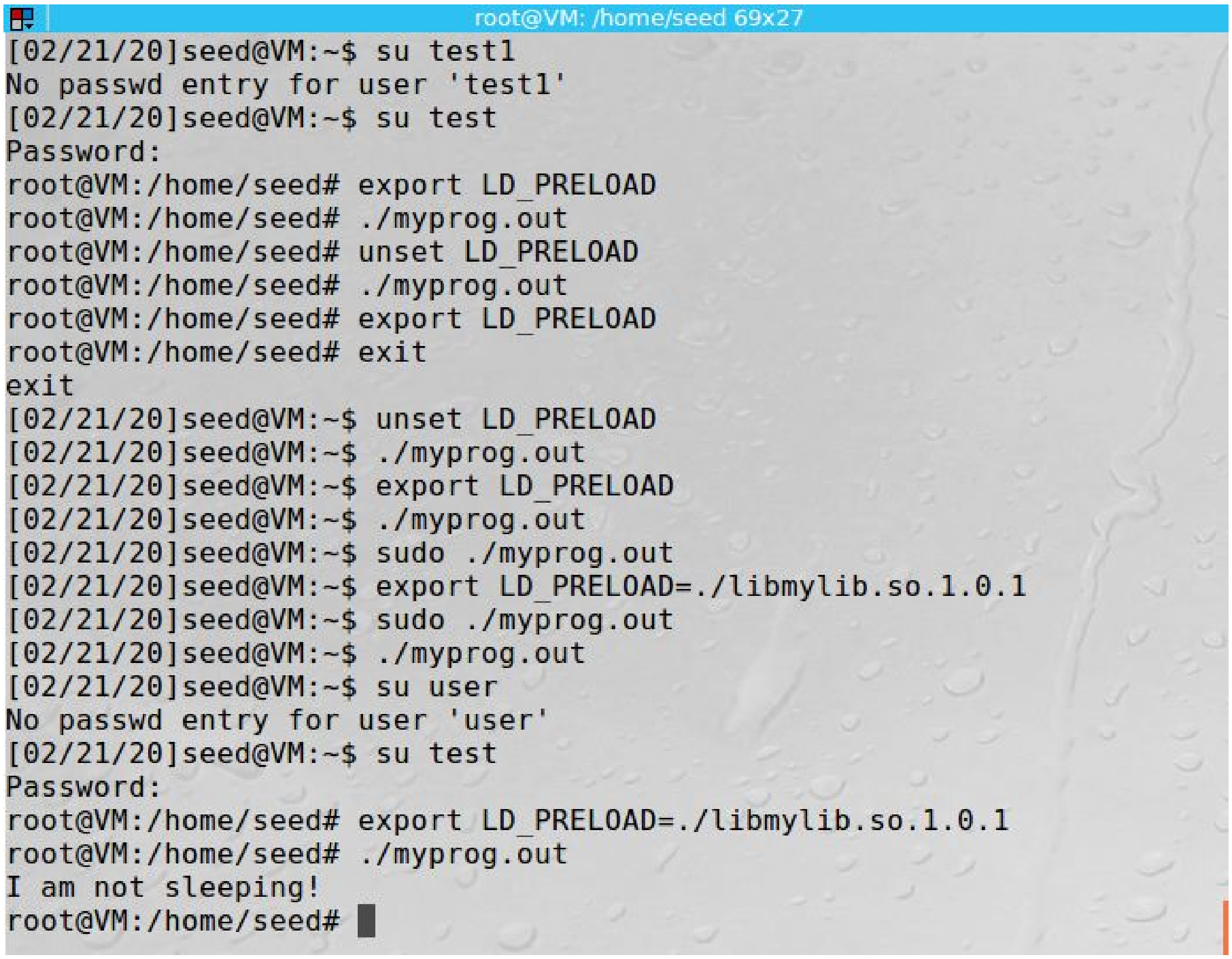
Analysis: It looks like my code runs either way when I call the program which shows the specified export PATH. I even tested with another user and the program worked without root privilege, so I believe that is because Set-UID gives us root privileges during execution of the program.

Section 2.7

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

Description: See how Set-UID programs deal with environment variables where we can affect a dynamic loader/linker (LD_PRELOAD).

Evidence:



```
root@VM: /home/seed 69x27
[02/21/20]seed@VM:~$ su test1
No passwd entry for user 'test1'
[02/21/20]seed@VM:~$ su test
Password:
root@VM:/home/seed# export LD_PRELOAD
root@VM:/home/seed# ./myprog.out
root@VM:/home/seed# unset LD_PRELOAD
root@VM:/home/seed# ./myprog.out
root@VM:/home/seed# export LD_PRELOAD
root@VM:/home/seed# exit
exit
[02/21/20]seed@VM:~$ unset LD_PRELOAD
[02/21/20]seed@VM:~$ ./myprog.out
[02/21/20]seed@VM:~$ export LD_PRELOAD
[02/21/20]seed@VM:~$ ./myprog.out
[02/21/20]seed@VM:~$ sudo ./myprog.out
[02/21/20]seed@VM:~$ export LD_PRELOAD=./libmylib.so.1.0.1
[02/21/20]seed@VM:~$ sudo ./myprog.out
[02/21/20]seed@VM:~$ ./myprog.out
[02/21/20]seed@VM:~$ su user
No passwd entry for user 'user'
[02/21/20]seed@VM:~$ su test
Password:
root@VM:/home/seed# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed# ./myprog.out
I am not sleeping!
root@VM:/home/seed#
```

Analysis: It appears that exporting the library as root caused the normal sleep to occur. Then when we exit from root, the normal seed user had our version of sleep. When we used user1 and exported, our version of sleep was called. I think the main reasons why other users who tried to use the function but didn't get the same as the root is because the LD_* environment variable is not inherited by the children process. Thus each user that is not the root has to export LD_PRELOAD…

Section 2.8

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

Description: We are given a scenario that uses system and execve which are quite dangerous if it's used in privileged programs such as Set-UID programs. We explore both systems and execve statements.

Evidence:

```
/bin/bash 62x23
[02/21/20]seed@VM:~$ sudo nano myprog2.1.c
[sudo] password for seed:
[02/21/20]seed@VM:~$ sudo chown root myprog2.c
[02/21/20]seed@VM:~$ sudo chmod 4755 myprog2.c
[02/21/20]seed@VM:~$ sudo chown root myprog2.1.c
[02/21/20]seed@VM:~$ sudo chmod 4755 myprog2.1.c
[02/21/20]seed@VM:~$ cat myprog2.1.c
#include <string.h>
#include <stdio.h>
#include <stdlib.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];
```

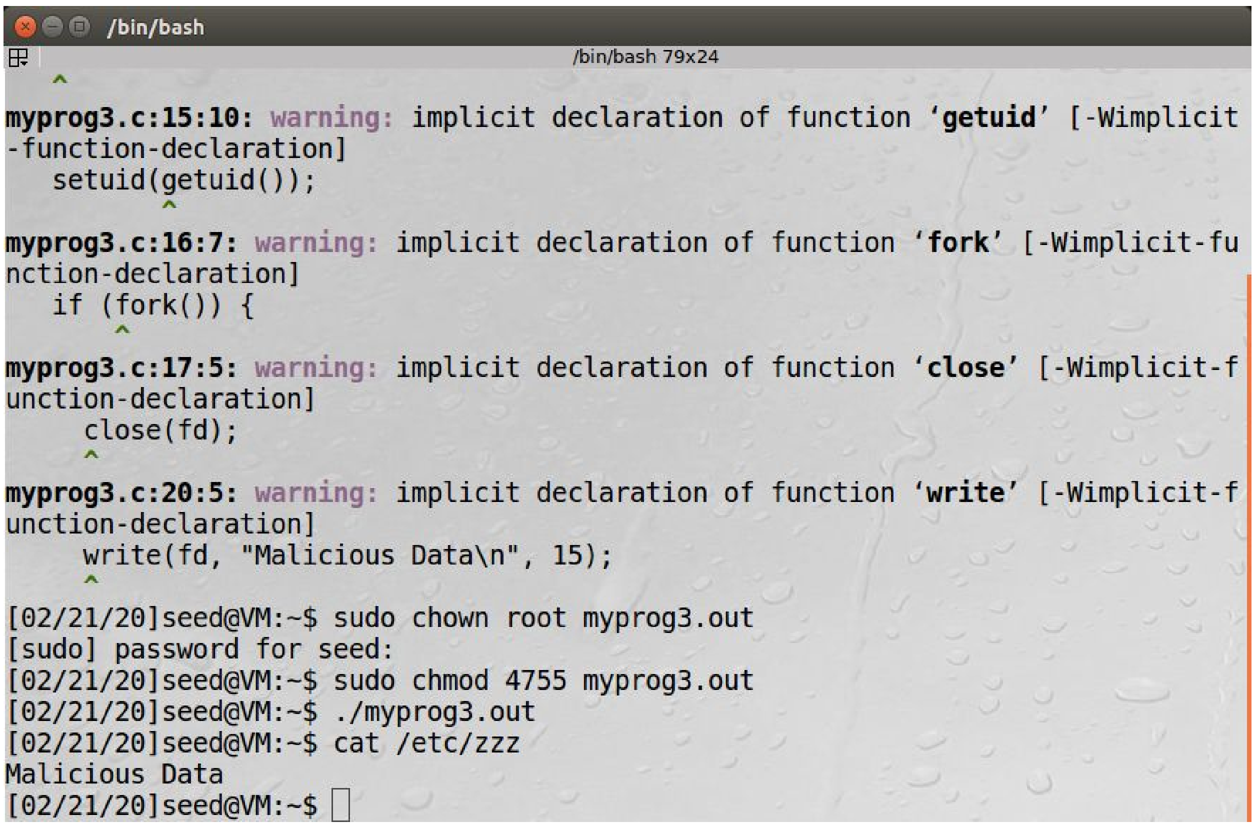
Analysis: It appears that since both system and execve had Set-UID privileges they are able to call use root and do commands, however since system calls shell to execute, system is even more vulnerable to shell commands which you can delete files since you have root access. So, although execve is safer than shell, both with Set-UID privileges of the root user can post security risks, especially system calls.

Section 2.9

Task 9: Capability Leaking

Description: Apply Principle of Least Privilege using Set-UID , but if you don’t use setuid properly then there is a capable leaking issue where they gain privileges when they shouldn’t. This is most likely caused when not setting uid before any actions occur which introduces a possible leak of privileges.

Evidence:



```
/bin/bash
/bin/bash 79x24
    ^
myprog3.c:15:10: warning: implicit declaration of function ‘getuid’ [-Wimplicit
-function-declaration]
   setuid(getuid());
          ^
myprog3.c:16:7: warning: implicit declaration of function ‘fork’ [-Wimplicit-fu
nction-declaration]
   if (fork()) {
       ^
myprog3.c:17:5: warning: implicit declaration of function ‘close’ [-Wimplicit-f
unction-declaration]
     close(fd);
     ^
myprog3.c:20:5: warning: implicit declaration of function ‘write’ [-Wimplicit-f
unction-declaration]
     write(fd, "Malicious Data\n", 15);
     ^
[02/21/20]seed@VM:~$ sudo chown root myprog3.out
[sudo] password for seed:
[02/21/20]seed@VM:~$ sudo chmod 4755 myprog3.out
[02/21/20]seed@VM:~$ ./myprog3.out
[02/21/20]seed@VM:~$ cat /etc/zzz
Malicious Data
[02/21/20]seed@VM:~$
```

Reflection

After completing this lab, I have a better understanding on how environment variables can affect programs and how there are some security risks to environment variables that you must be cautious about. Additionally, I learned how to do Set-UID for programs while also acknowledging the possible danger of poorly handled programs or code that can abuse the Set-UID.   Lastly, I learned how to apply Least Privileged which can still be a security hazard if not handled properly. At the end of the day, it seems as though that we have a lot of safety mechanisms but the user must still be cautious and think about what he is doing with those safe mechanisms, otherwise those safety mechanisms are basically useless and can be a vulnerability to the system.

Ethan Booker

Race Condition Vulnerability Lab

2/21/2020

Description of the overall goals of the lab

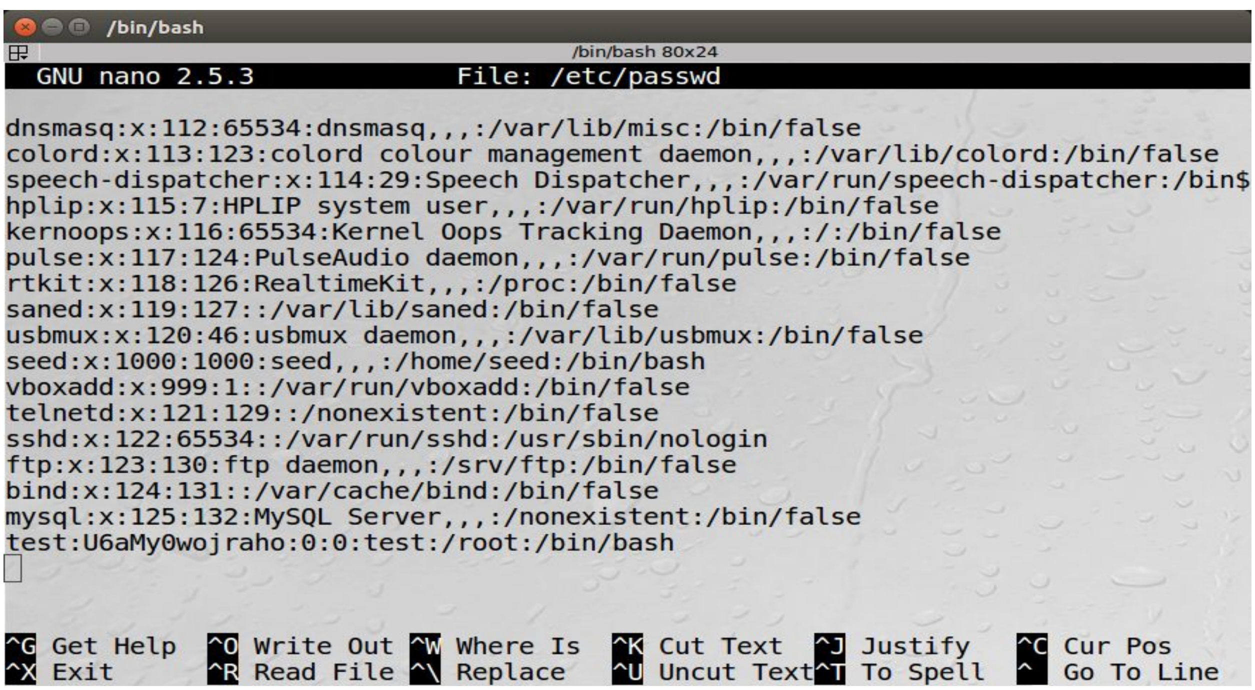
The overall goal of this lab is to gain hands-on experience of exploiting a race-condition vulnerability to change the behavior of a program, in this case it's gaining root privilege. After learning how to exploit programs using a race-condition, the lab will also help teach us ways to defend against this type of exploit.

Section 2.3

Task 1: Choosing Our Target

Description: Use the application to exploit our target which is the /etc/passwd file to gain root access. Basically, manipulate the file to change the password field to our desired value.

Evidence:



Analysis: By replacing the “x” after the name of host name with Ubuntu’s magic value, you can essentially create an account that doesn’t need a password but you have to press enter. This is a security mechanism by Ubuntu since if you didn’t replace the x then people who are naive would not know what the password is unless they looked in the /etc/shadow file for the password.

Section 2.4

Task 2: Launching the Race Condition Attack

Description: An application of the race condition in action, where the vulnerable Set-UID program will try to point a file to our target file using the critical window between access() and fopen() calls.

Evidence:

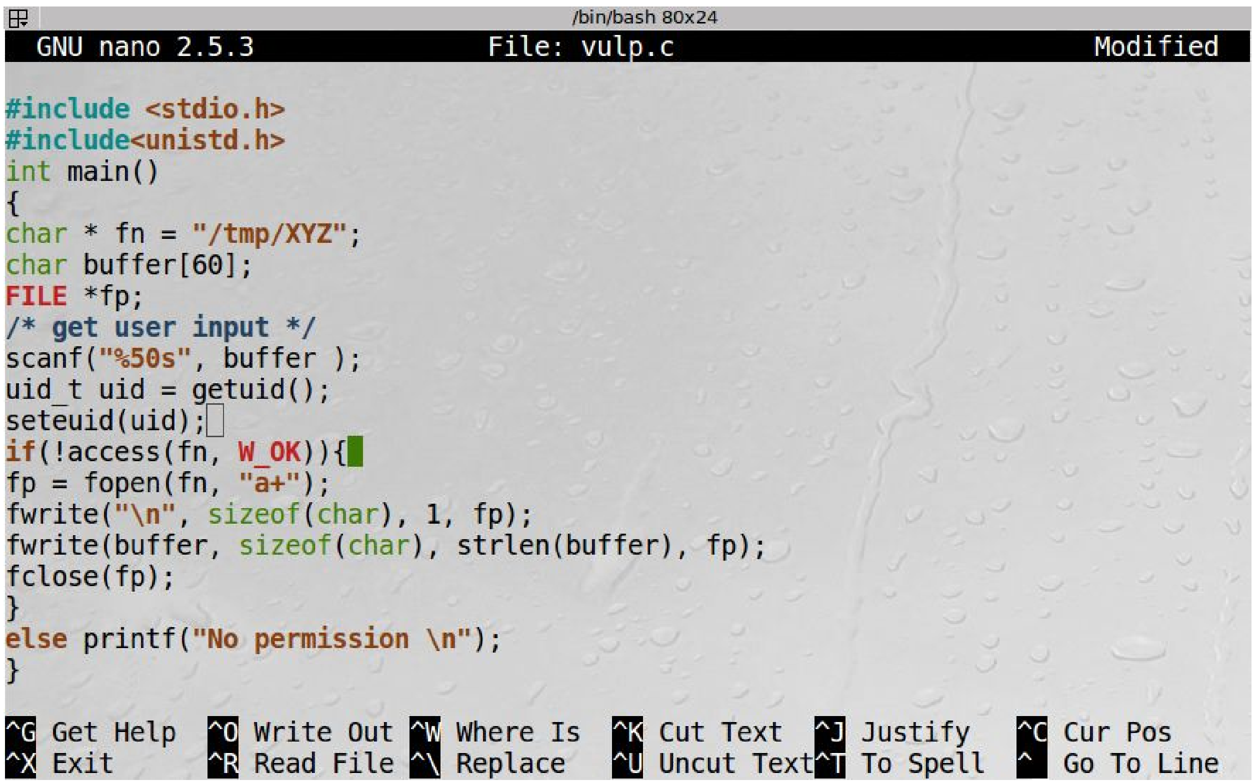
Analysis: Since this vulnerability tries to take advantage of a critical time window, timing is key and due to us not being able to get perfect timing, this is a probabilistic strategy. I spent countless hours running a script that would run this program with no luck. The overall goal is to take advantage of a race condition in order to gain linkage in order to get root privilege of our target.

Section 2.5

Task 3: Countermeasure: Applying the Principle of Least Privilege

Description: Try to apply principle of least privilege on the program.

Evidence:



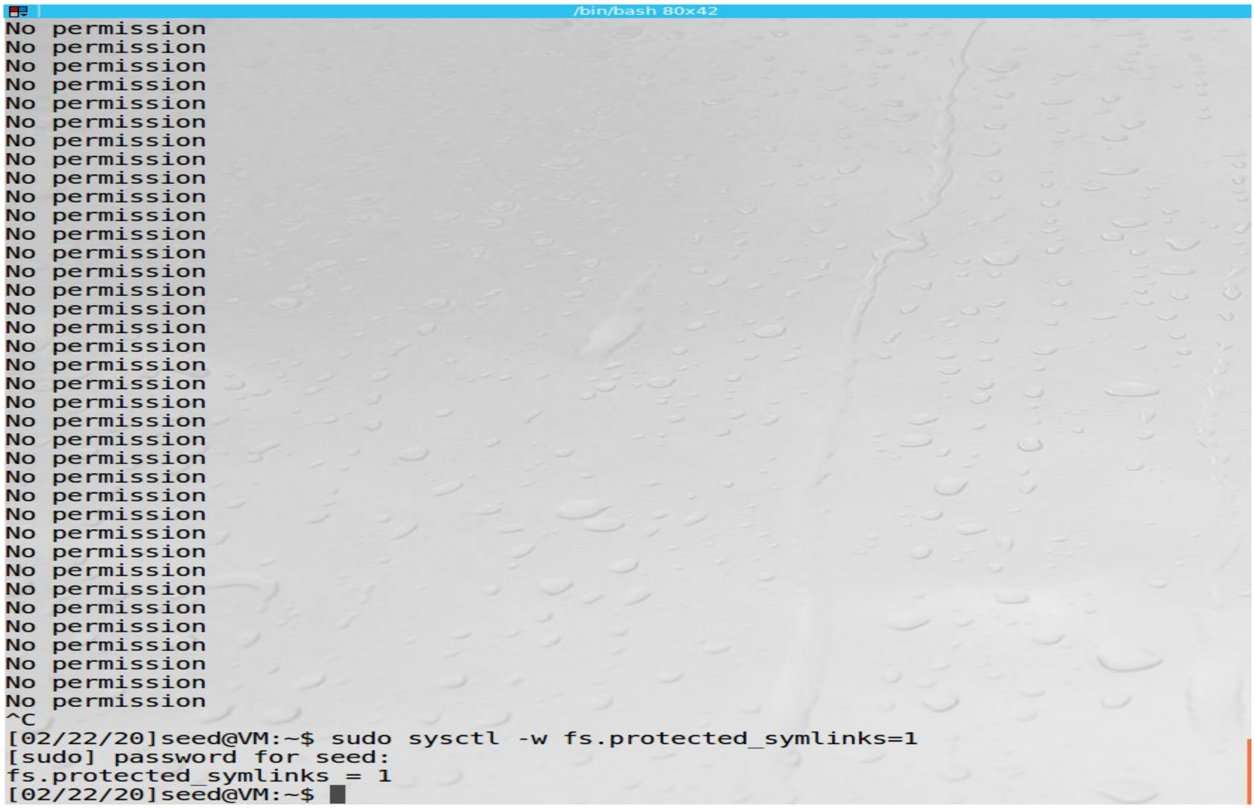
Analysis: Since my program didn't breach the vulnerability, I can't tell for sure, but I assume that based on the current user, principle of least privilege using seteuid will check to make sure this program can only work if the user has access to that file. This could prevent the race condition but it would open up possibilities of improper use of seteuid. If this file isn't set properly, someone could modify the file to allow the condition to occur again.

Section 2.6

Task 4: Countermeasure: Using Ubuntu's Built-in Scheme

Description: Turn back on Ubuntu's built-in protection scheme.

Evidence:



```
/bin/bash 80x42
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
^C
[02/22/20]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=1
[sudo] password for seed:
fs.protected_symlinks = 1
[02/22/20]seed@VM:~$
```

Analysis: Clearly this has to work due to the fact that we had to take off the symlink in order to do this lab. A symbolic link is basically a file that contains references of either a relative/absolute path to another file/directory. A symlink can be an alias for the target and trying to remove a file that's in the symbolic link will only remove the link in the symbolic link. Symlinks do need maintenance as there can be dangling targets if they were moved and not updated in the link. Another downside of symlinks is that they change the file system from a tree to a directed graph causing issues of navigation and need programs that directly handle symbolic links to manipulate them.

- When the link is deleted, the target remains unchanged.
- When the target is moved, the link becomes invalid.
- Supports relative pathing, crossing filesystem boundaries, and for both files and directories for Unix systems.
- This is pretty much a typical symlink can do besides what I mentioned earlier.