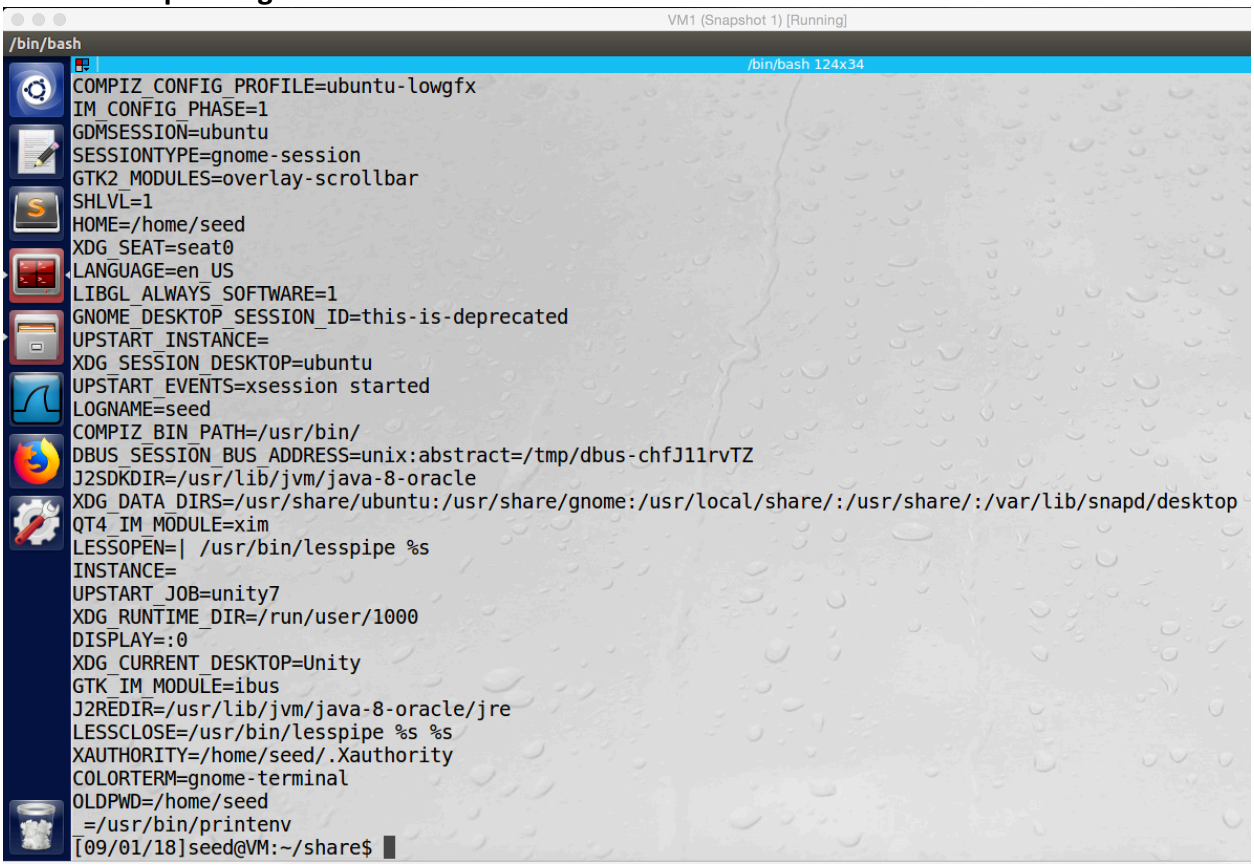
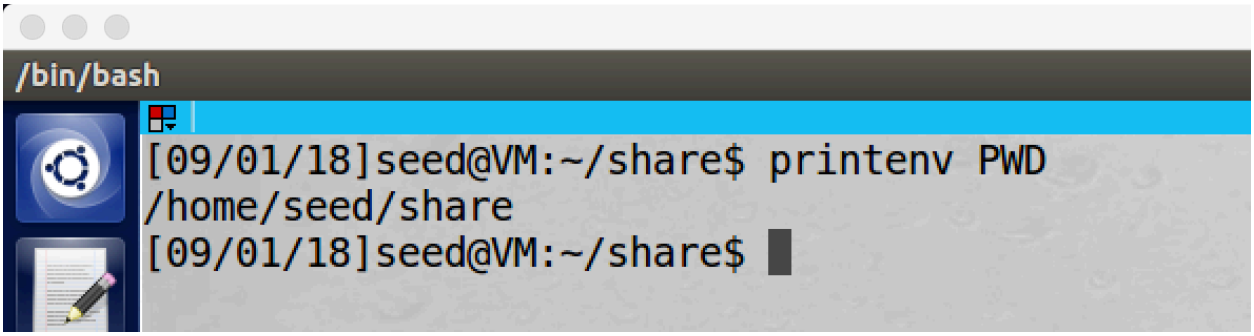
CSE643 Lab1
Yishi Lu
09/11/2018

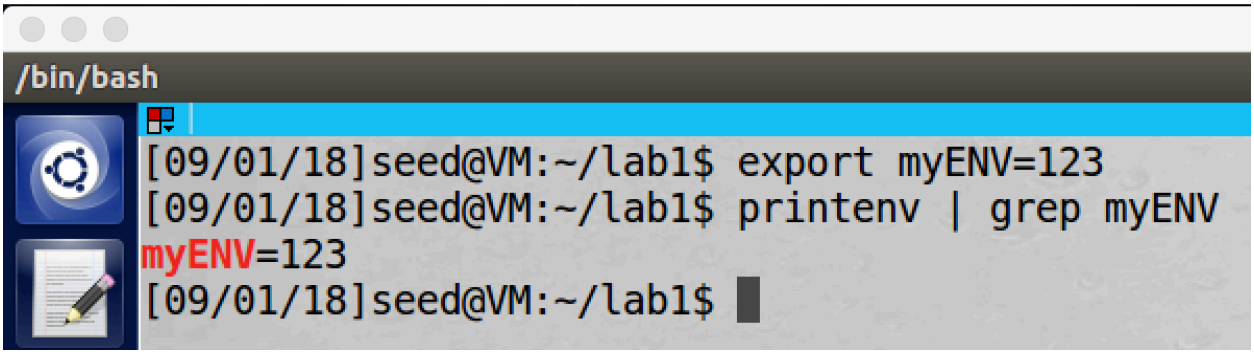
Task1: Manipulating Environment Variables



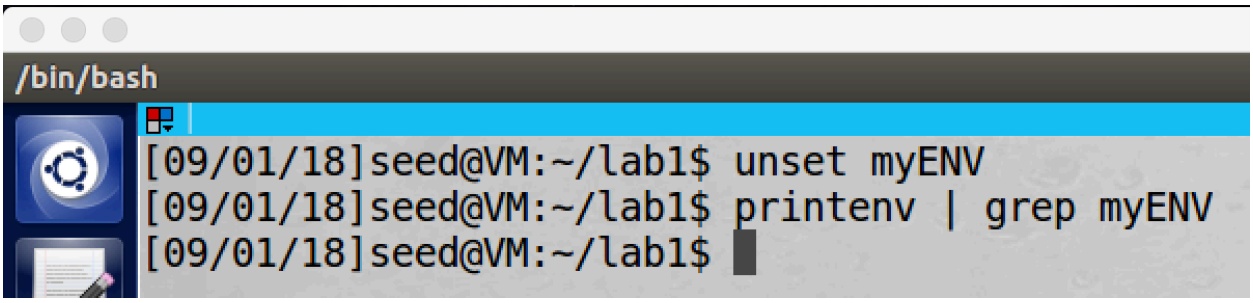
screenshot1, we run “printenv” command to print environment variables of current process.



screenshot2, we run “printenv PWD” to print the value of environment variable PWD



screenshot3, we run command “export” to export our own environment variable myENV to be 123; and then we run printenv again to print the value of myENV which is 123



screenshot4, we run command “unset” to unset environment variable myENV; and then we run printenv again to print the value, but this time there is no such variable

Observation and Explanation:

we first run “printenv” command, and this command prints all environment variables of current shell program (As screenshot1 shows).

And then we run “printenv PWD” command to get the value of environment variable PWD, PWD contains the full path name of the current working directory. So by this command, it prints the current working directory which is “/home/seed/share” (As screenshot2 shows).

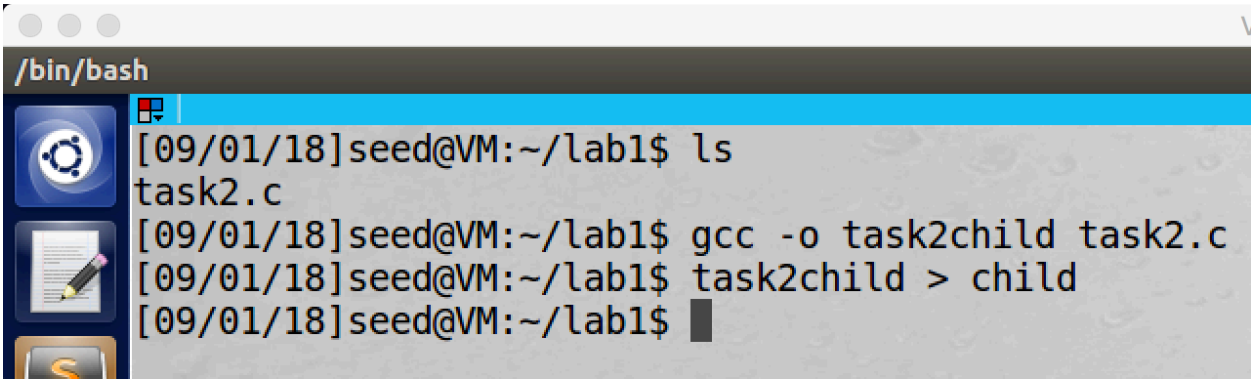
In the next step, we use command “export” to set our own environment variable myENV to be 123. As the screenshot3 shows, the myENV value is 123.

Finally, we use command “unset” to unset environment variable myENV. As the screenshot4 shows, after we run the command, there is no such environment variable anymore.

Task2: Passing Environment Variable from Parent Process to Child Process

Step1:

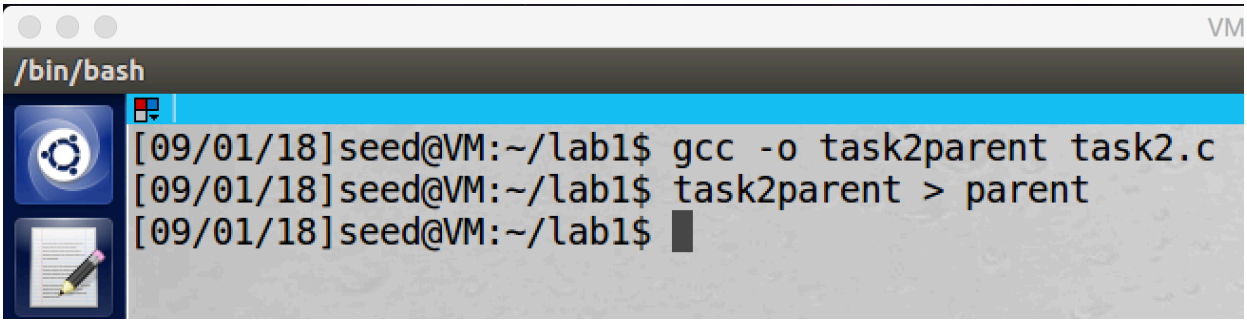
screenshot1, copying the code from lab description to the VM. Here I comment printenv() in parent process, and uncomment printenv() in child process.



screenshot2, I compile and run the program, and I save the output into file child

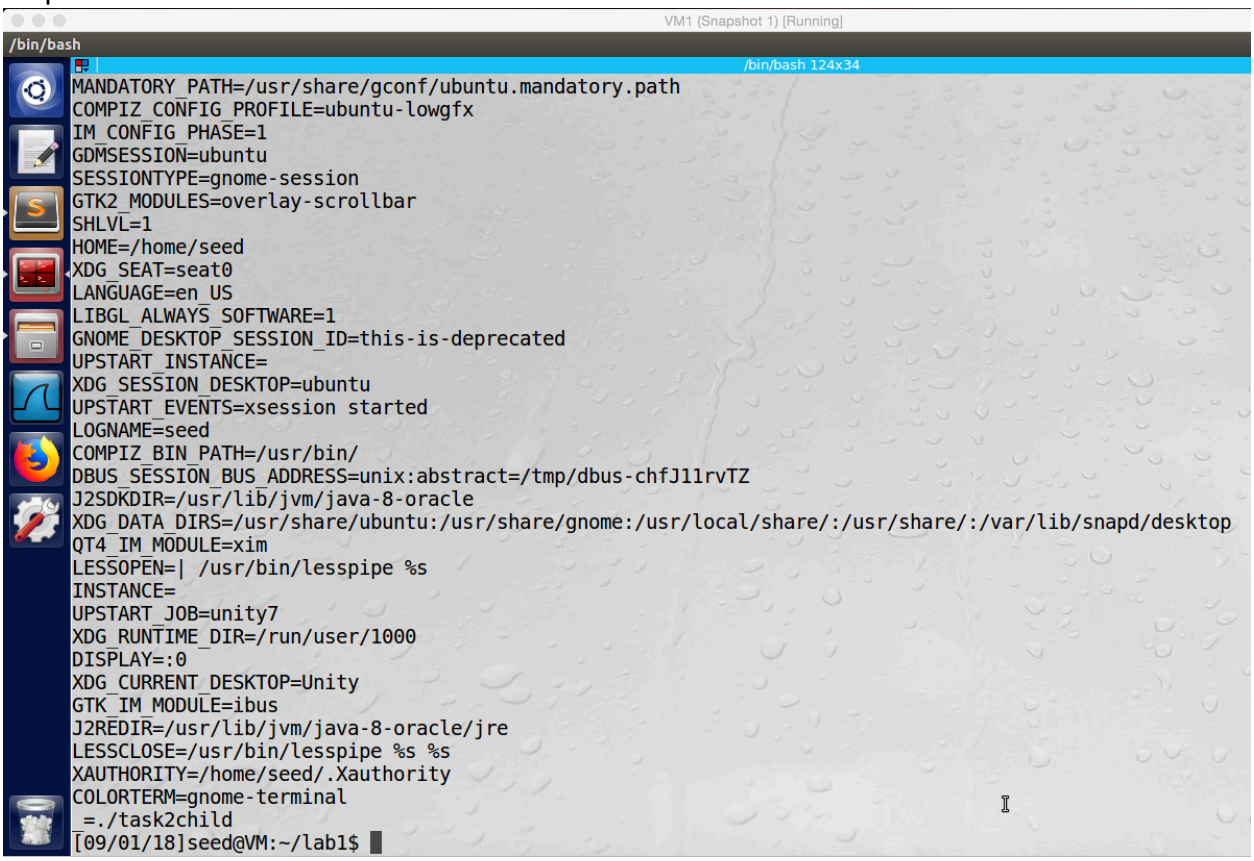
Step2:

screenshot3, this time I comment printenv() in child process, but uncomment printenv() in parent process

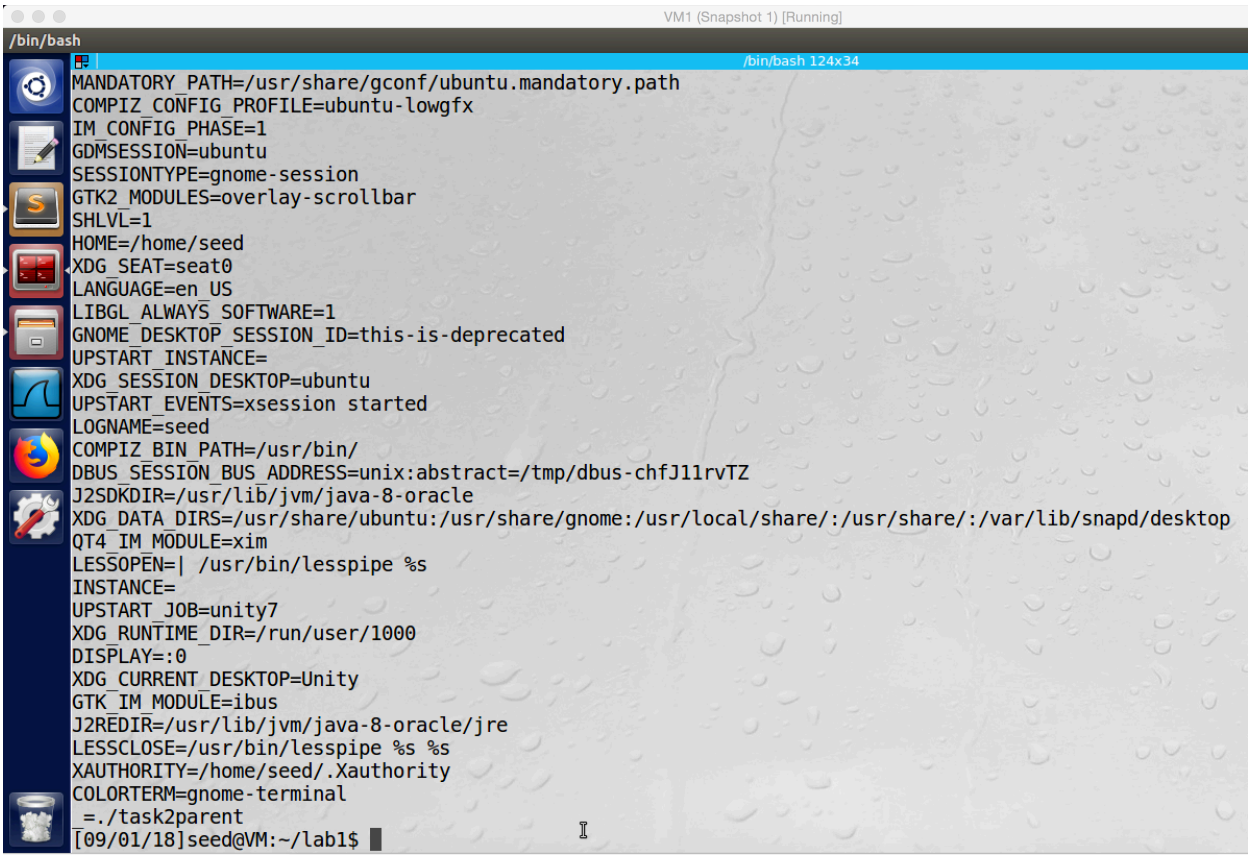


screenshot4, I compile and run the program, and I save the output into another file which called parent

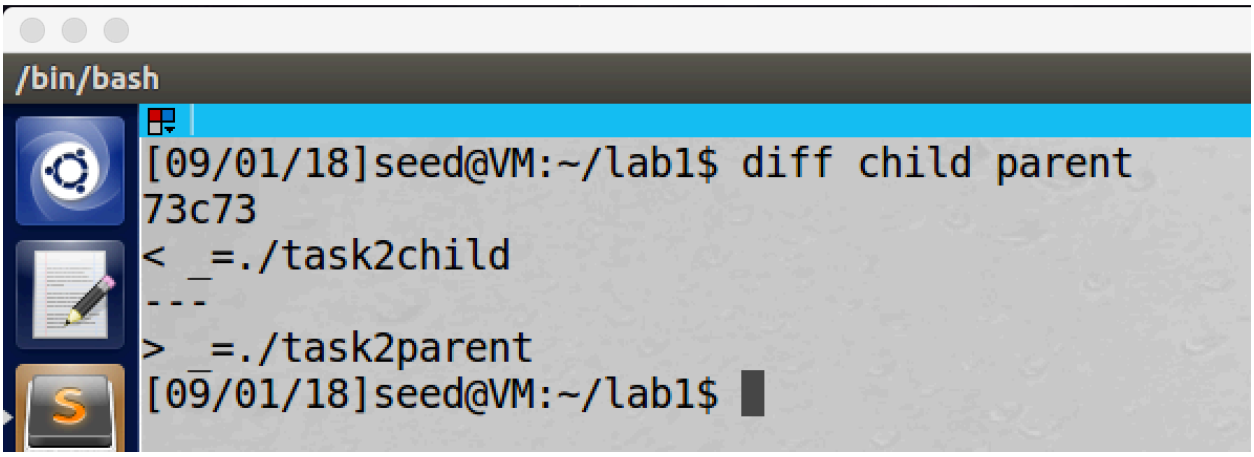
Step3:



screenshot5, I first run the file taks2child without saving it to another file, and it print all environment variables of the child process.



screenshot6, and then I also run the file task2parent without saving to other file, and it prints out all environment variables of the parent process.



screentshot7, I use diff command to compare child file and parent file. In fact, the only difference is the file name, so the child process inherits all environment variable from the parent process.

Observation and Explanation:

In first step, the program will print all environment variables of the child process. Because there are many string, we save the output to another file which called “child” (screenshot 1 and 2).

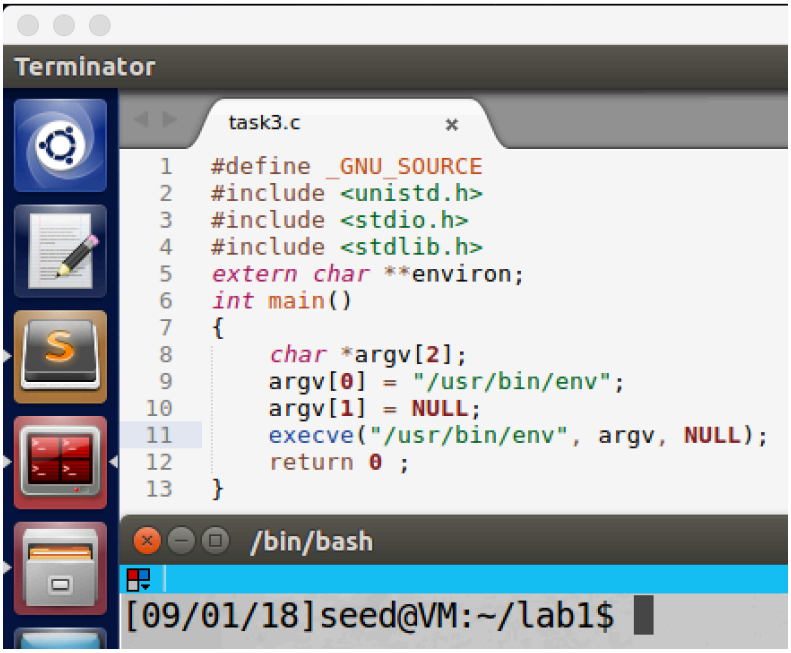
In step2, we change the program little bit, so this time the program will print environment variables of the parent process. Because the same reason, we also save the output to another file which called “parent” (screenshot 3 and 4).

In step3, we firstly run these two compiled files: task2child and task2parent. As screenshot 5 and 6 show, the outputs are very similar, most values are same. Because there are too many string, it is very hard to see the difference between these two outputs. Therefore, we should compare the “child” file and “parent” file to see the difference. As screenshot7 shows, we use command diff to compare child file and parent file. As the result shows, they are basically the same.

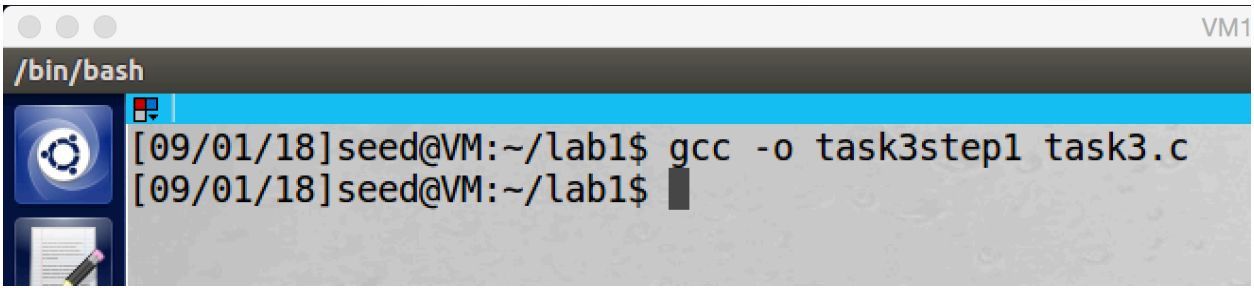
In conclusion, when we use fork() to create a child process, the child process’s memory is duplicate of the parent’s memory. Therefore, the child process inherits all environment variable from the parent process.

Task3: Environment Variables and execve()

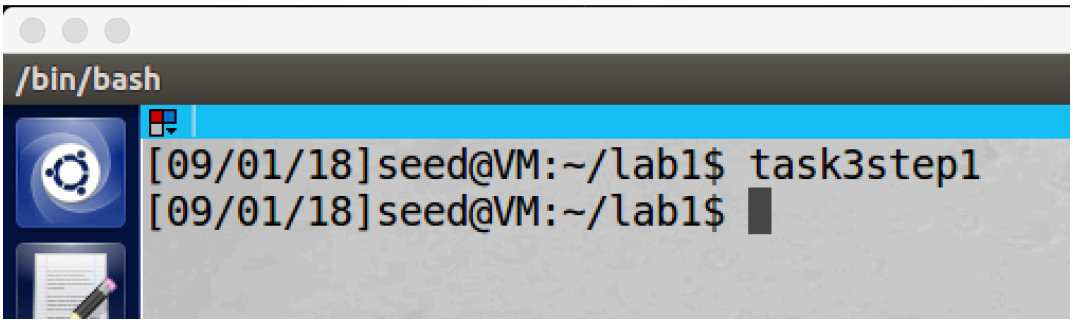
Step1:



screenshot1, copying code for this task from lab description. At this time we pass NULL as third parameter to execve()



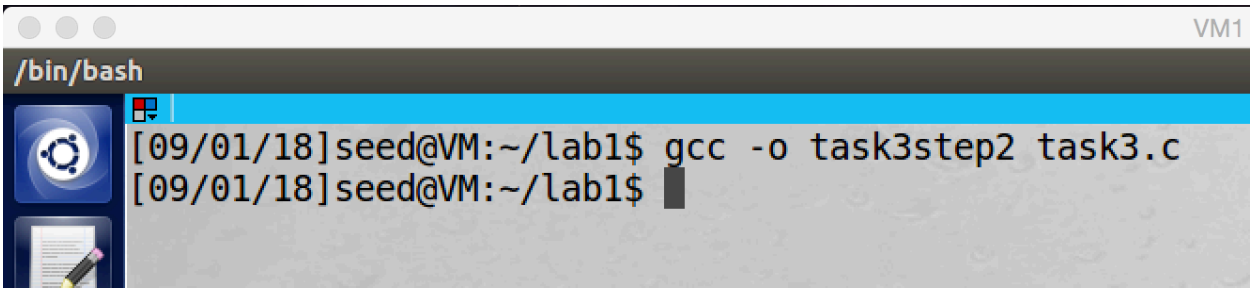
screenshot2, compile the program



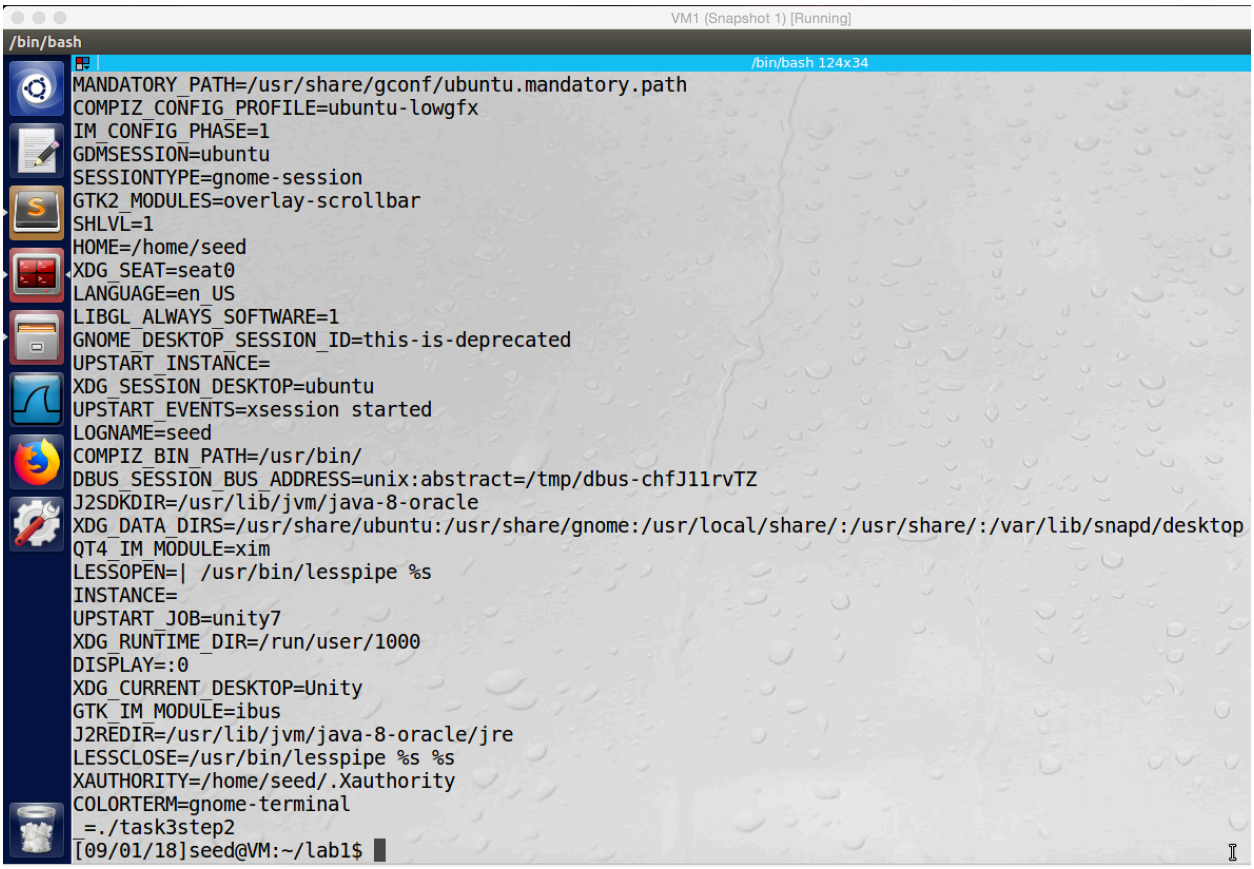
screenshot3, after I run the compiled program, nothing is printed.

Step2:

screenshot4, I change the program a little bit, I changed the NULL to environ. Environ is a pointer which point to the environment variables of the current process, which means we pass the current process's environment variables to the new program



screenshot5, I recompile the program.



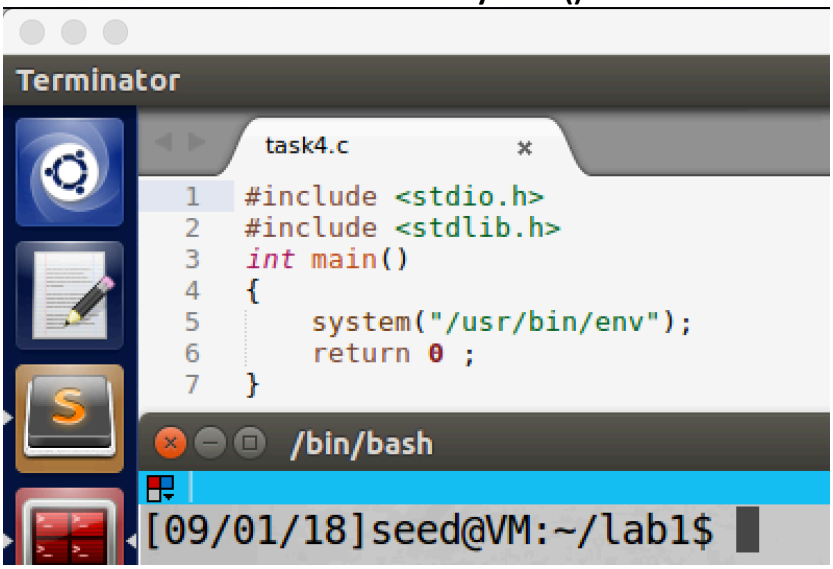
screenshot6, after I run the program, a lot of environment variables are printed.

Observation and Explanation:

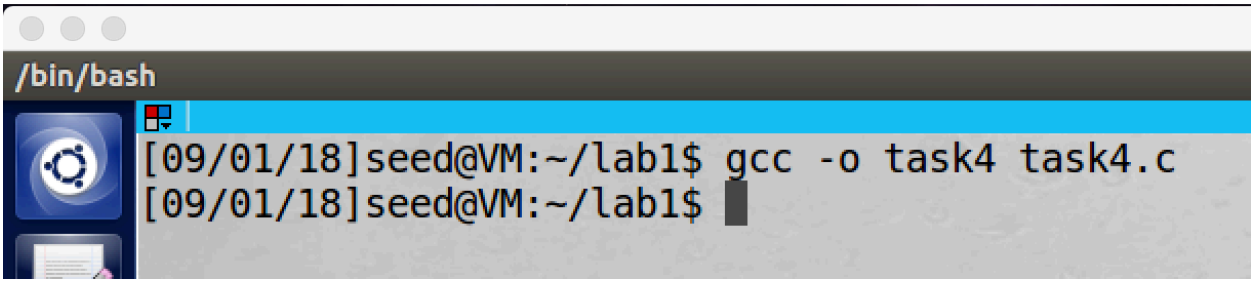
In this task, we run a simple program, which uses system call execve() to run another program-/usr/bin/env. In step1, we pass NULL as third parameter into function execve(). After we run the compiled program, we see that the program (task3step1) does not print anything, which means the process does not have any environment variables (screenshot 1, 2, and 3). In step2, we pass environ as third parameter into the system call execve() (screenshot 4 and 5), environ is a pointer which point to the environment variables of the current process, so we actually pass the current process's environment variables to the new program. When we run the compiled program (task3step2), it prints a lot of environment variables (screenshot 6). And these environment variables are coming from the calling process.

Step3 (in conclusion), when a process use system call execve() to run a new program, the system call will overwrite the current process's memory by the data provided by the new program. The third parameter of execve() decides which environment variables are passed to the new program. In step1, we pass NULL as environment variable to the system call, so all environment variables of current process are lost, and NULL is passed to the new program as environment variables. As step2 shows, when we pass the current process's environment variables to the system call execve(), these variables are passed to the new program.

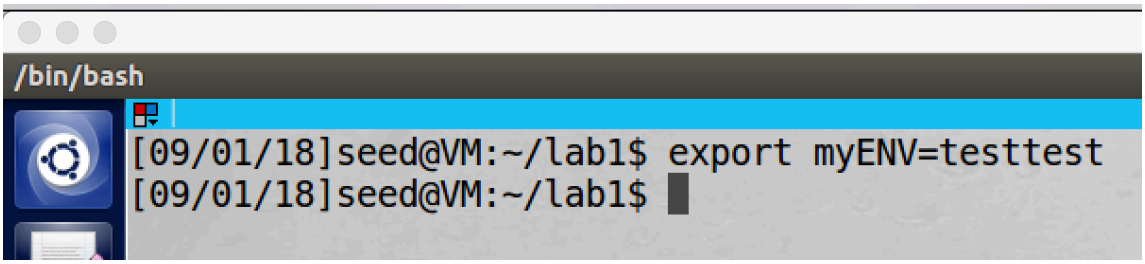
Task4: Environment Variables and system ()



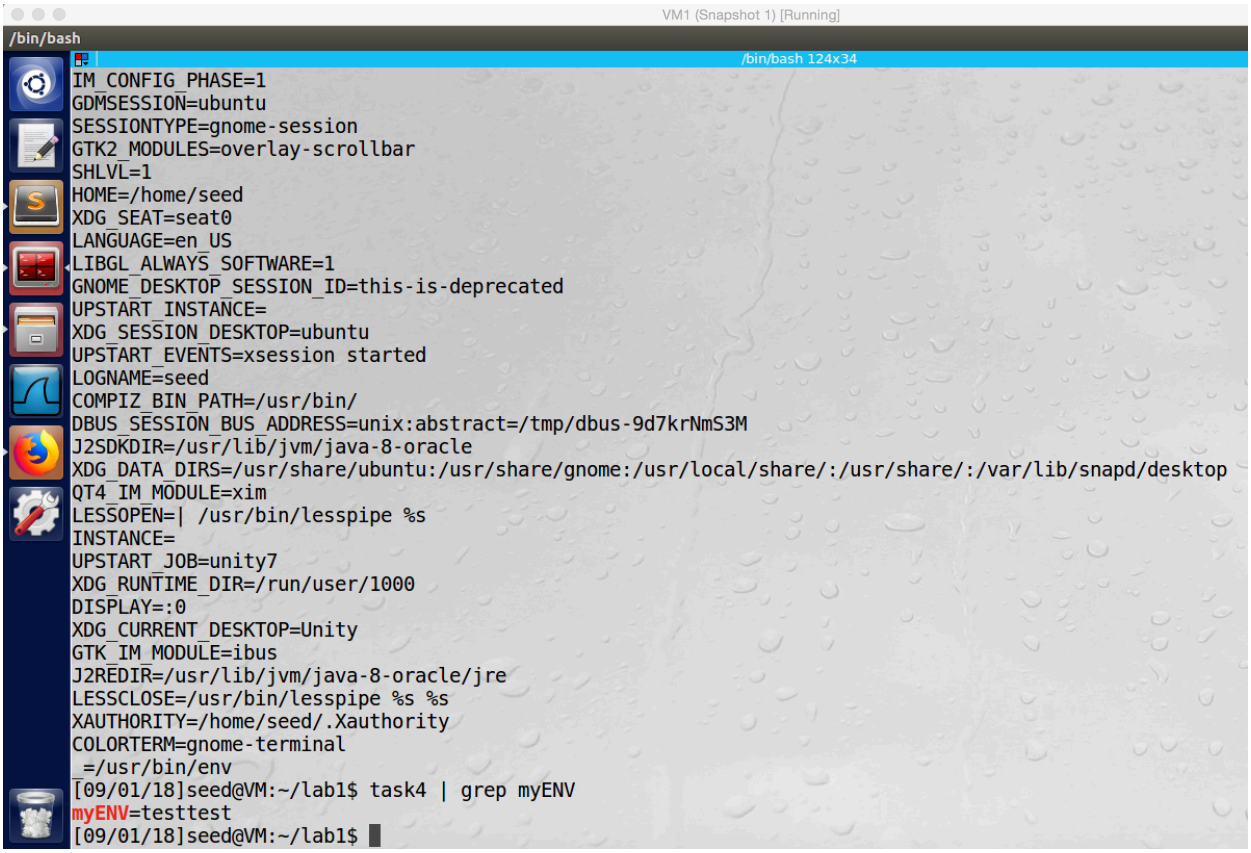
screenshot1, copying the program from lab description



screenshot2, compile the program



screenshot3, we export our own environment variable myENV=testtest



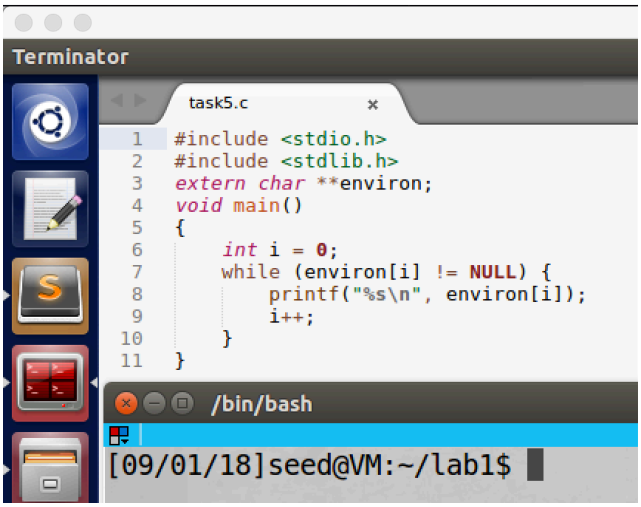
screenshot4, after we run the program, a lot of environment variable are printed. And we can see our environment variable myENV is printed out as well. Which means the environment variables of the calling process is passed to the /bin/sh.

Observation and Explanation:

We first copy the program from the lab description, and then we compile and run it (screenshot 1 and 2). And then we export our own environment variable myENV=testtest (screenshot3). Finally, we run the program, and we see myENV is printed out (screenshot4). In fact, when the program runs, the system() does not run the external program directly, instead it will call excel(), and excel() will call execve() and pass to it the environment variables; then execve() will execute the shell program /bin/sh, and the shell program will execute the external program. To verify this, we export our own environment variable myENV; after we run the program, we see that myENV is printed out, which means environment variables of the calling process is passed to the /bin/sh program.

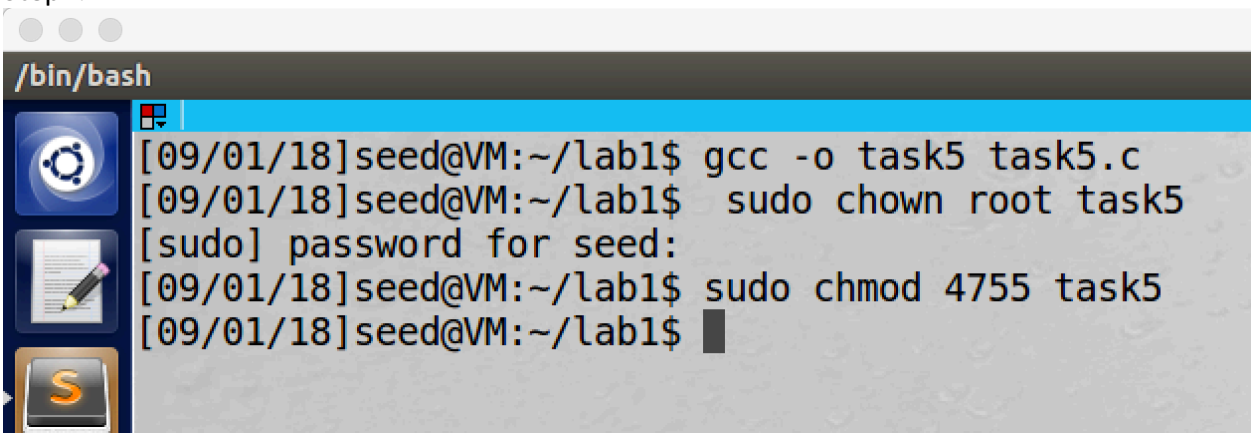
Task5: Environment Variable and Set-UID Program

Step1:



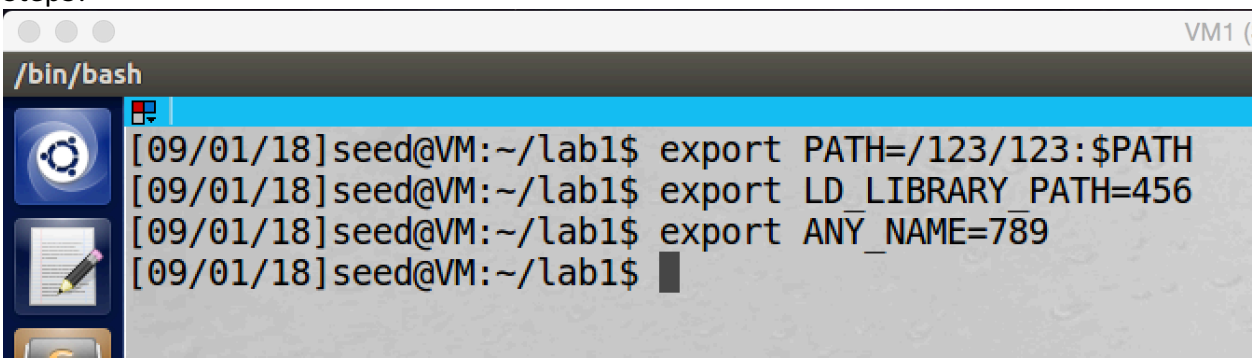
screenshot1, copying the program from lab description

Step2:

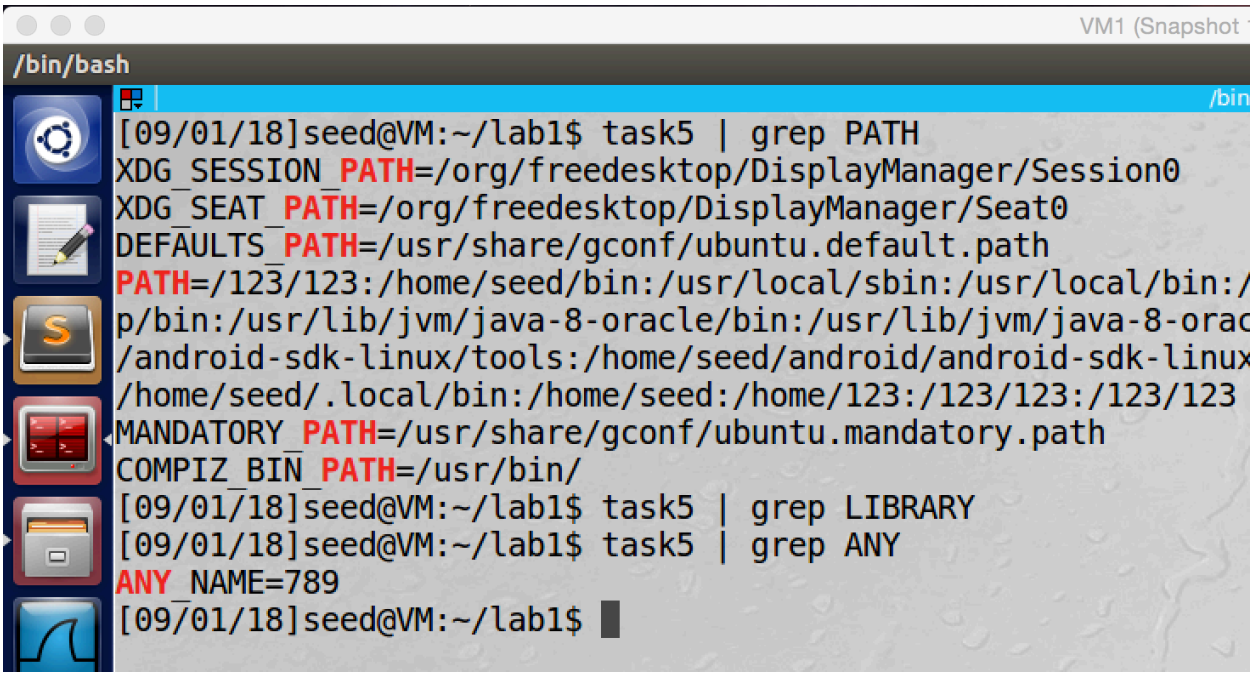


screenshot2, compile the program, and then we make it to be a Set-UID root program

Step3:



screenshot3, using export command to change values of the above three environment variables.



screenshot4, when we run the program, we can see that PATH and ANY_NAME environment variables are got into the child process. But the LD_LIBRARY_PATH does not.

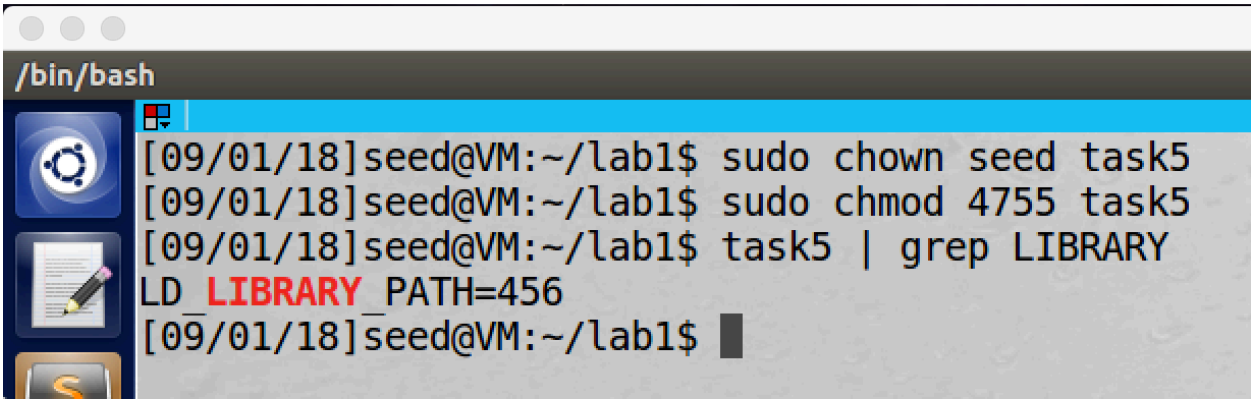
Observation and Explanation:

In step1, we just copy and compile the program which is from lab description.

In step2, we make the compiled program to be a Set-UID root program

In step3, we changed values of three environment variables, we changed PATH to $PATH:/123/123, LD_LIBRARY_PATH to 456, and ANY_NAME to 789 (screenshot 3). Then we run the program, and we see the value of PATH and ANY_NAME are modified by us, but the LD_LIBRARY_PATH does not appear in the output (screenshot4).

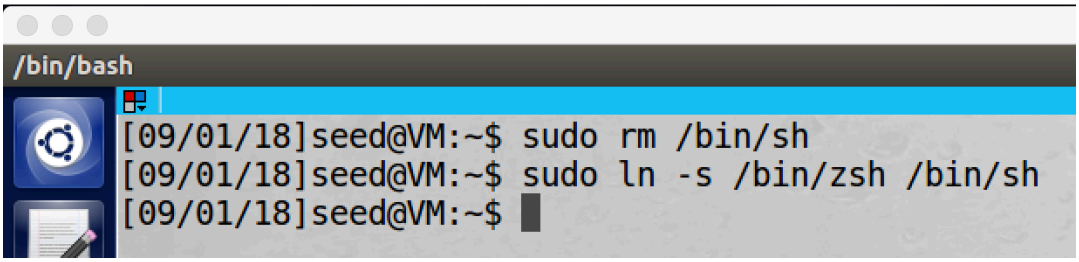
Actually, I expect all those three variables should be passed to the child process, but the LD_LIBRARY_PATH does not. So I make an experiment.



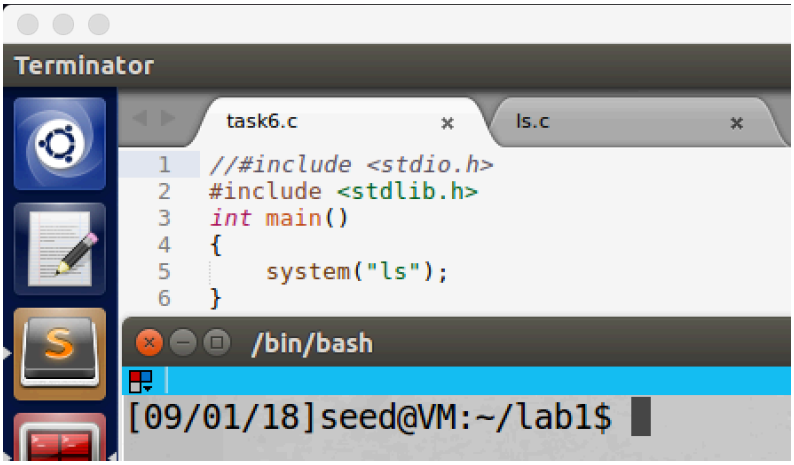
As the above screenshot shows, I change the program to be a Set-UID user program with user seed. And then I run the program again. At this time, we see the LD_LIBRARY_PATH is passed to the child process, and the value is 456 which we set before. The reason is, when a process's real and effective user IDs are different, the LD_LIBRARY_PATH environment variable

is ignored. Therefore, we cannot see it when our program is a Set-UID root program, and it is run by a normal user account (real user ID is seed, and effective user is root, they are different).

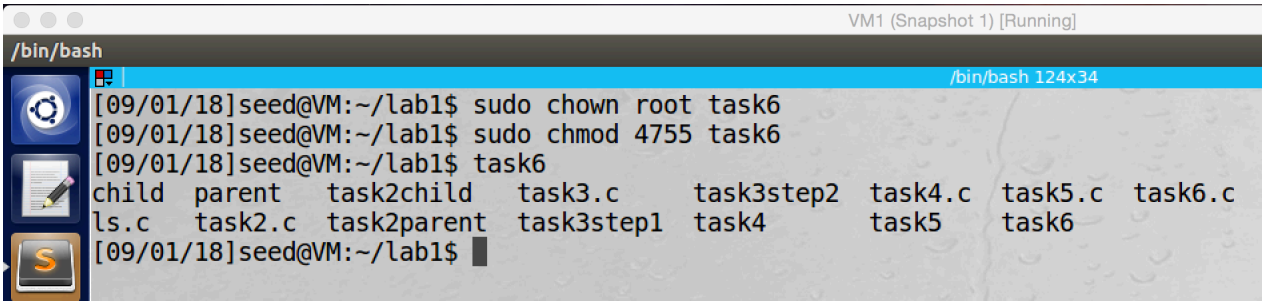
Task6: The PATH Environment Variable and Set-UID Programs



As the lab description says, before we do the task, we should run the above two commands to link the /bin/sh to bin/zsh because Ubuntu 16.04 countermeasure.

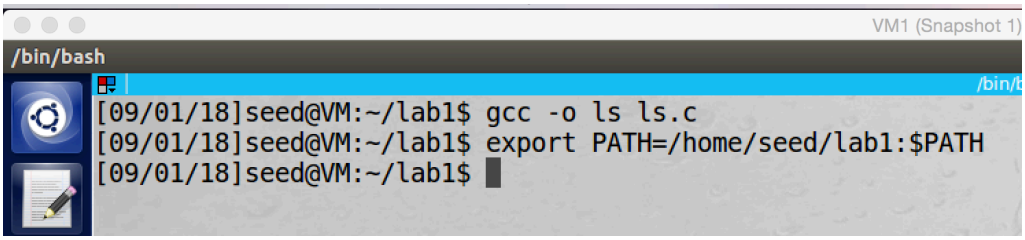


screenshot1, copying the program from lab description

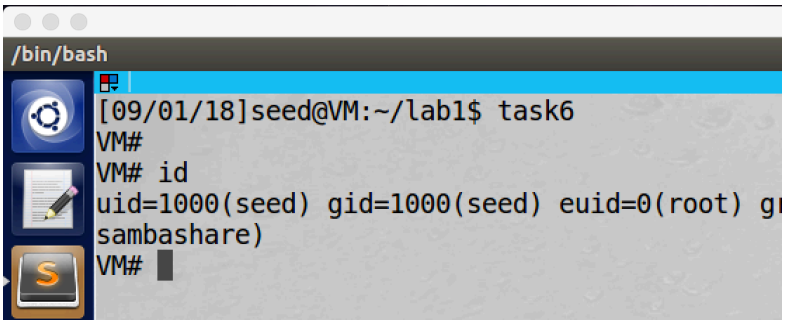


screenshot2, after compiling the program, we make it to be a Set-UID root program. When we run the program, it prints out the file in the current directory

screenshot3, creating my “ls” program which will get a new shell. If this command is run by root privilege, then we will get a root shell



screenshot4, we compile my ls program, and we set the PATH environment variable to be current directory (/home/seed/lab1).



screenshot5, finally we run the Set-UID program again, and this time our malicious ls program is executed. And we get a root shell (the euid is 0).

Observation and Explanation:

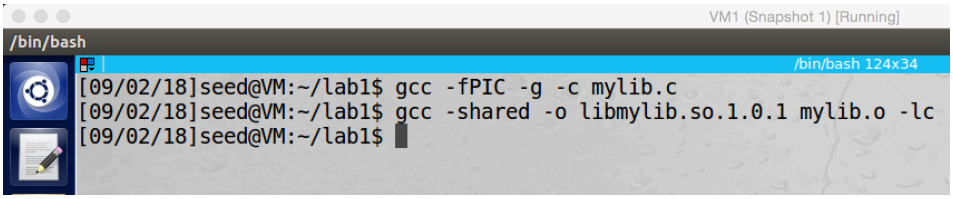
Actually, we can let the Set-UID program to run our malicious ls program. Because inside the program task6, we did not give the full file path of the command ls; instead we only give “ls”; so the system() will use environment variable PATH to find the program. If we change the environment variable PATH to the location of our malicious ls program, then our malicious ls program should be executed. Furthermore, inside our ls program, it runs “/bin/sh”, we can get root shell by this command if it is run by root privilege. We first, create a Set-UID program which will execute system(“ls”), such command will print all files under current directory (screenshot 1 and 2). Now we create and compile our malicious ls program (screenshot 3). And then we set the PATH environment variable to be PATH=/home/seed/lab1:$PATH, this is the

location of our malicious ls program (screenshot4). Finally, we run the Set-UID program again, at this time, we get a root shell. To verify this, we run id command, and we can see the euid is 0 (screenshot 5), which means this is a root shell. So our attack is successful, our malicious ls code is running with the root privilege.

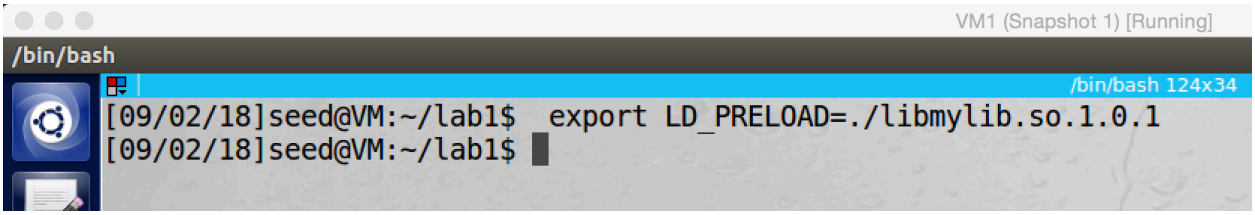
Task7: The LD_PRELOAD Environment Variable and Set-UID Programs

Step1:

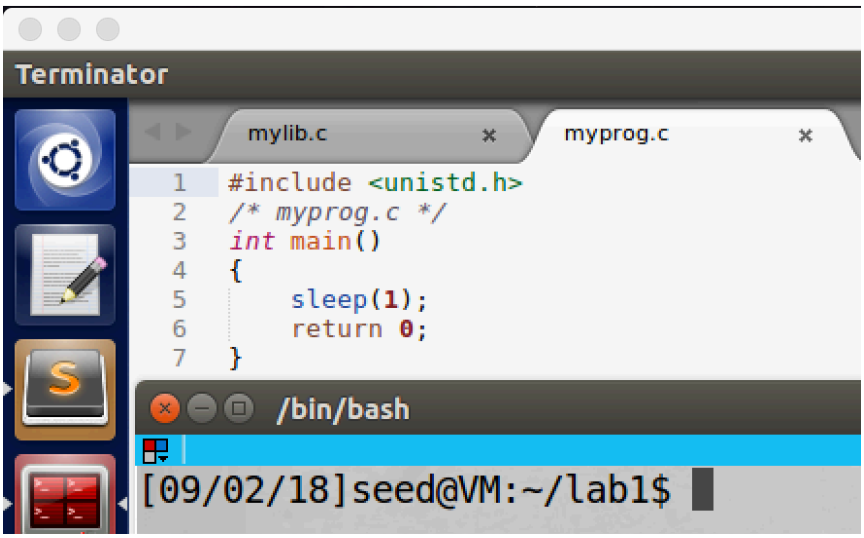
screenshot1, creating mylib.c



screenshot2, compile mylib.c



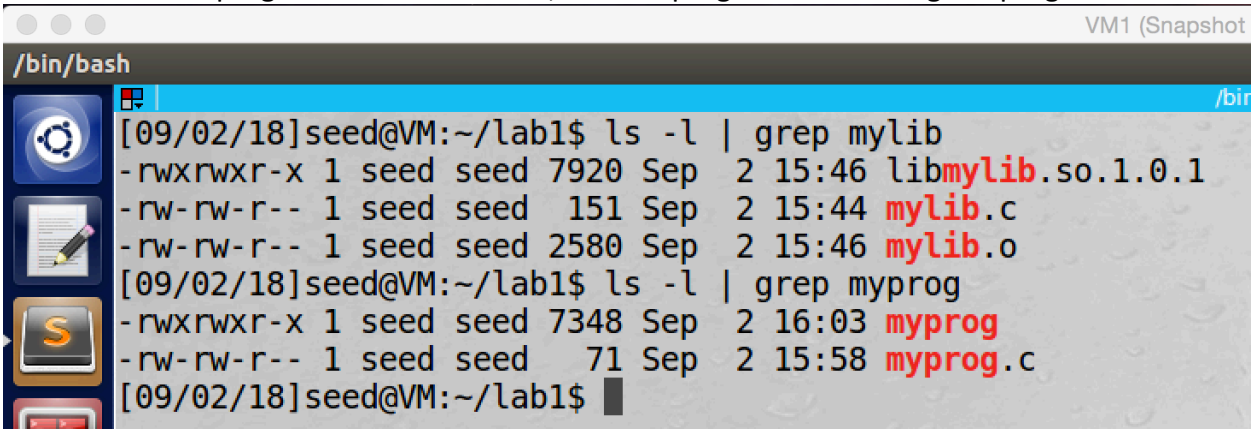
screenshot3, set the LD_PRELOAD environment variable



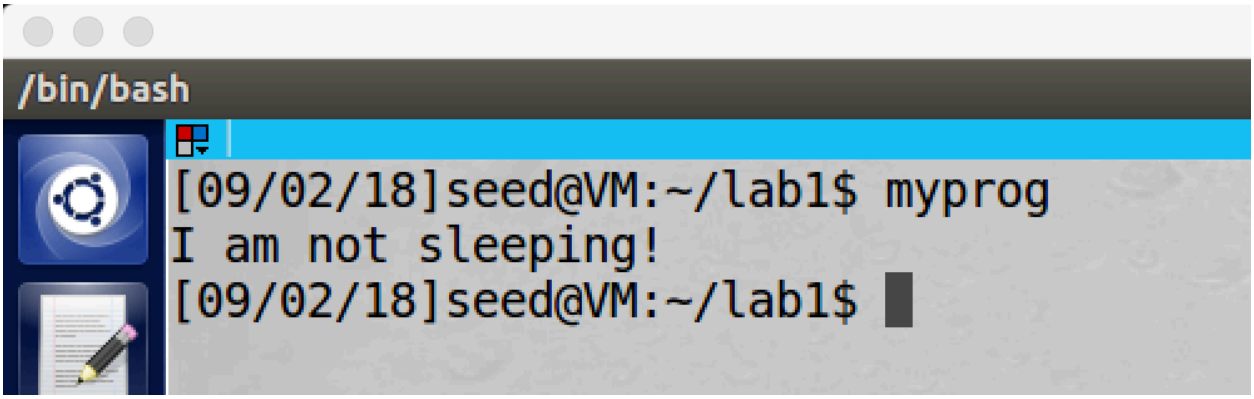
screenshot4, create my sleep program which called myprog.

Step2:

1. We first run the program as a normal user, and the program is also a regular program

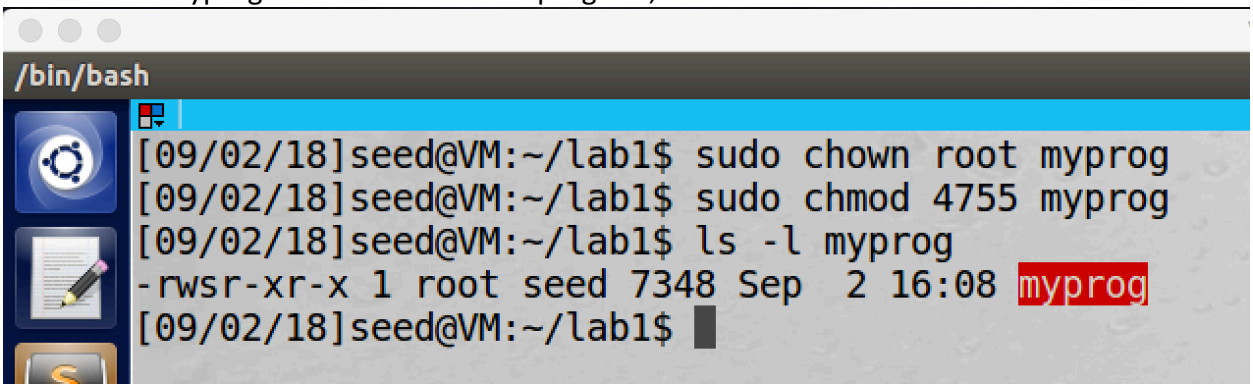


screenshot5, running myprog as a normal user, and the program is also a regular program

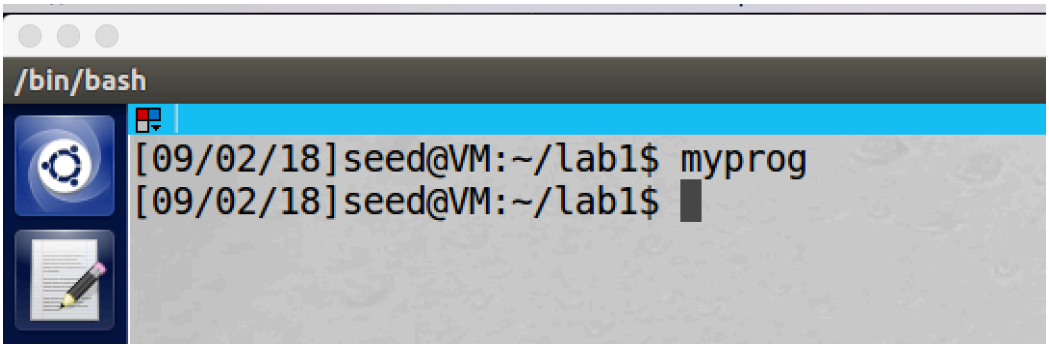


screenshot6, our sleep funcition is called instead of the one from libc, so our attack succeeds.

2. Now we set myprog to be a Set-UID root program, and run it as a normal user

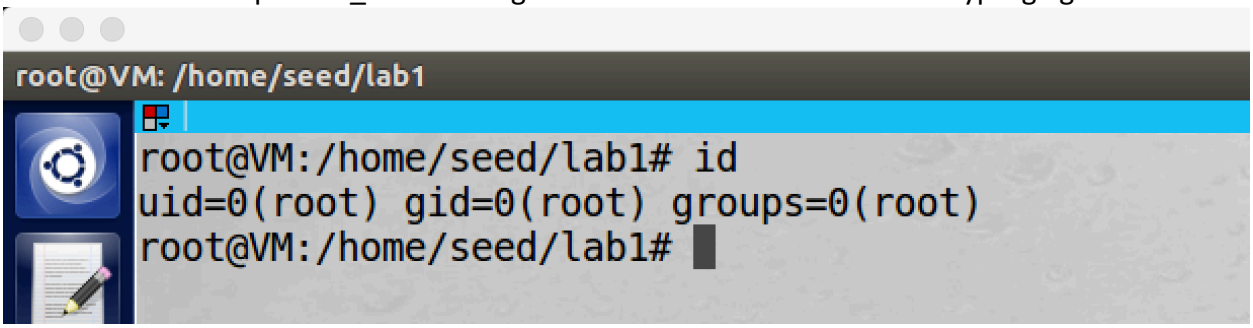


screenshot7, make myprog to be a Set-UID root program

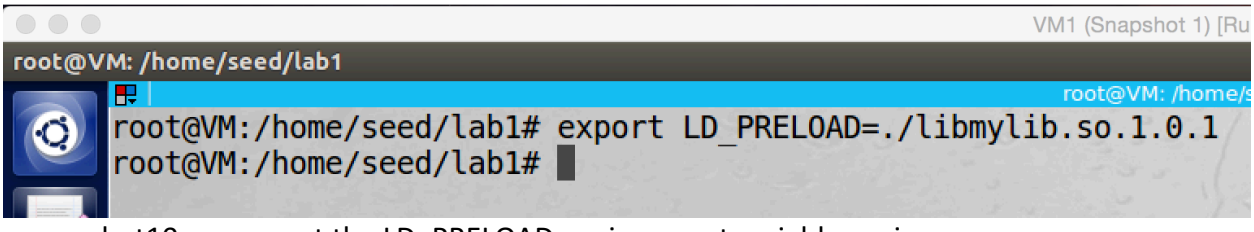


screenshot8, when we run myprog again, the sleep function from libc is called, so our attack fails.

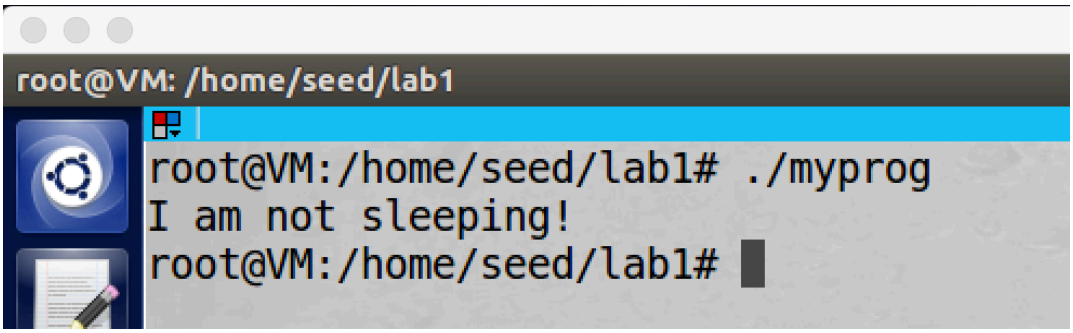
3. Now we want to export LD_PRELOAD again in the root account and run myprog again



screenshot9, we switch to a root account

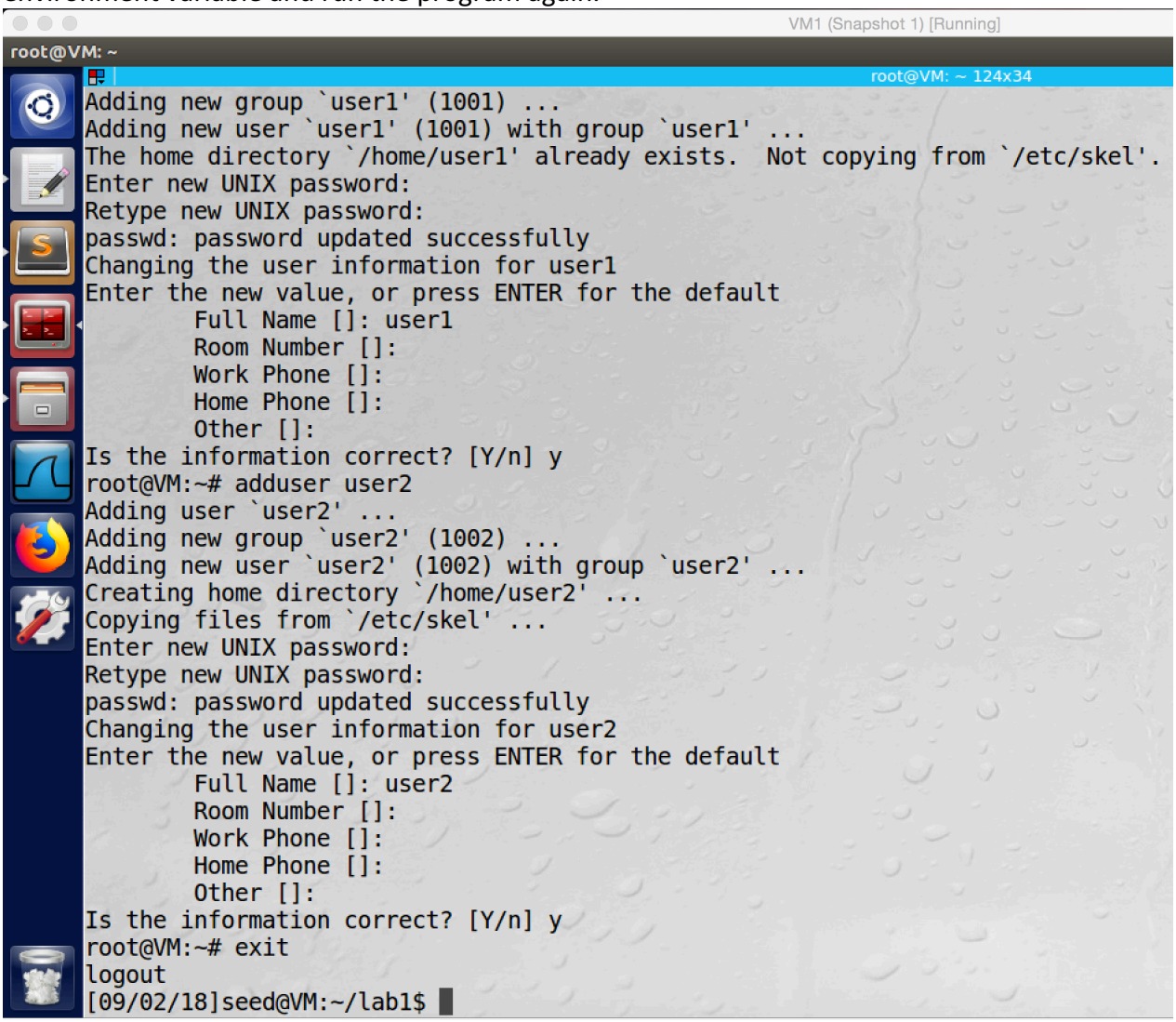


screenshot10, we export the LD_PRELOAD environment variable again

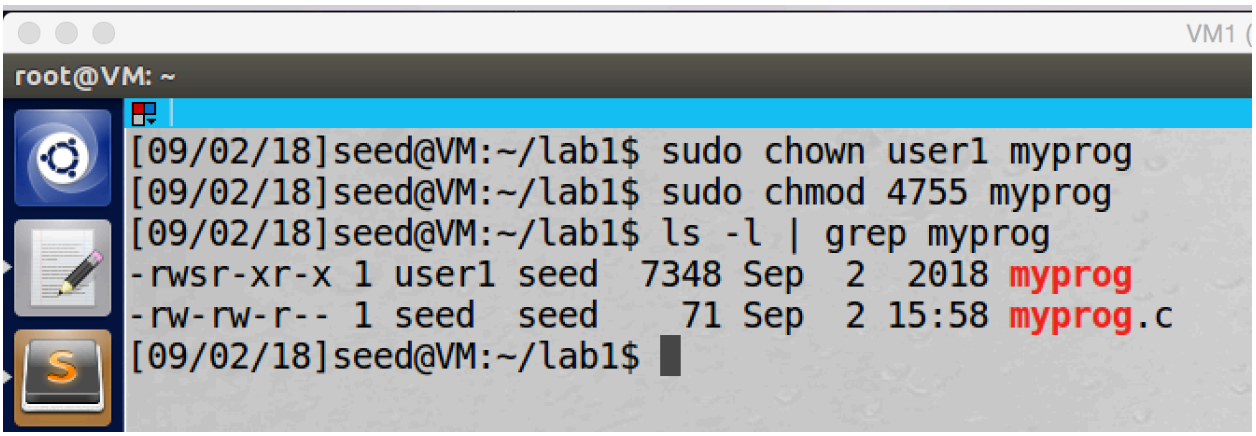


screesnhot11, we run the program again, at this time our sleep function is invoked, so our attack succeeds.

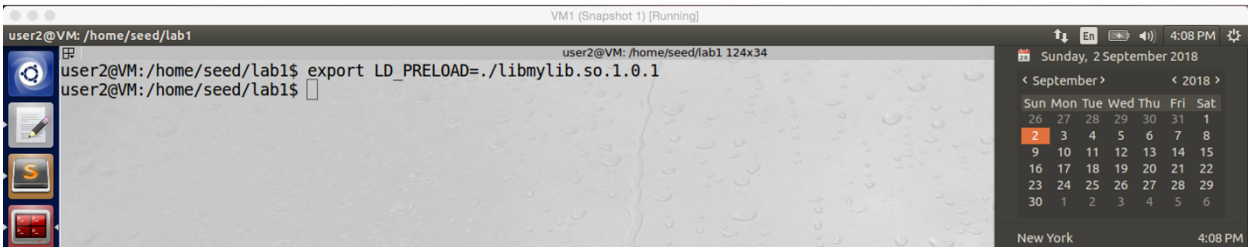
4. In this step we will create two new user accounts, user1 and user2. we first set myprog to be a Set-UID user1 program, and then we use user2 to export LD_PRELOAD environment variable and run the program again.



screenshot12, we create two new user account user1 and user2



screenshot13, we make myprog to be a Set-UID user1 program



screenshot14, we export LD_PRELOAD again by user2

screenshot15, we use user2 account to run the program, this time the sleep function from libc is called, so our attack fails.

Observation and Explanation (Step3):

In the first case, when myprog is a regular program, and we are normal users (screenshot5). Our sleep function is called (screenshot6), which means we successfully changed the LD_PRELOAD environment variable. Because we are a normal user, and the program is normal program, such attack cannot damage a lot to the system.

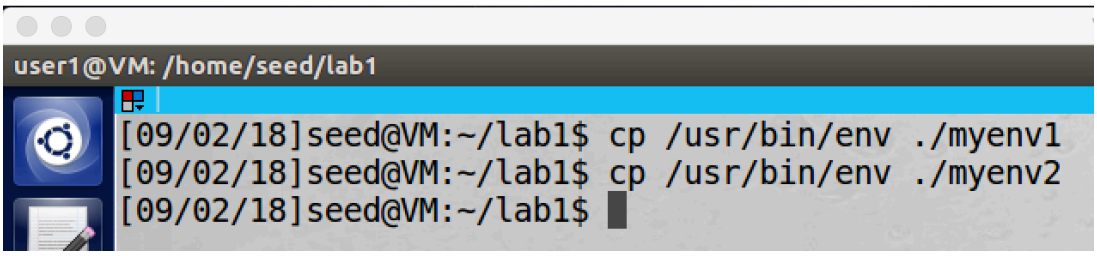
In second case, we make myprog to be a Set-UID root program (screenshot7), and then we run the program again by normal user account. But this time, the sleep function from libc is invoked, which means our attack fails (screenshot8). But I still do not know why, so let's continue to see more cases.

In third case, we set the program to be a Set-UID root program, and we export LD_PRELOAD and run the program as a root account (screenshot 9, 10, 11). This time, our sleep function is called again, which means our attack is successful.

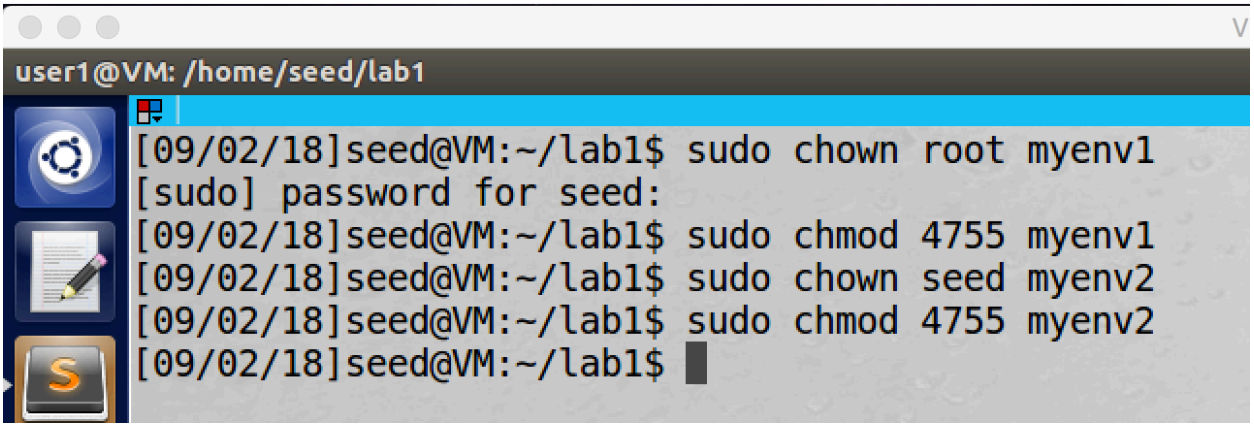
In fourth case, we first create two new user accounts, user1 and user2 (screenshot12). And then we set the program to be a Set-UID user1 program (screenshot 13). Afterwards, we export LD_PRELOAD and run the program by user2 (screenshot 14, 15). This time, the sleep function from libc is called, which means our attack fails.

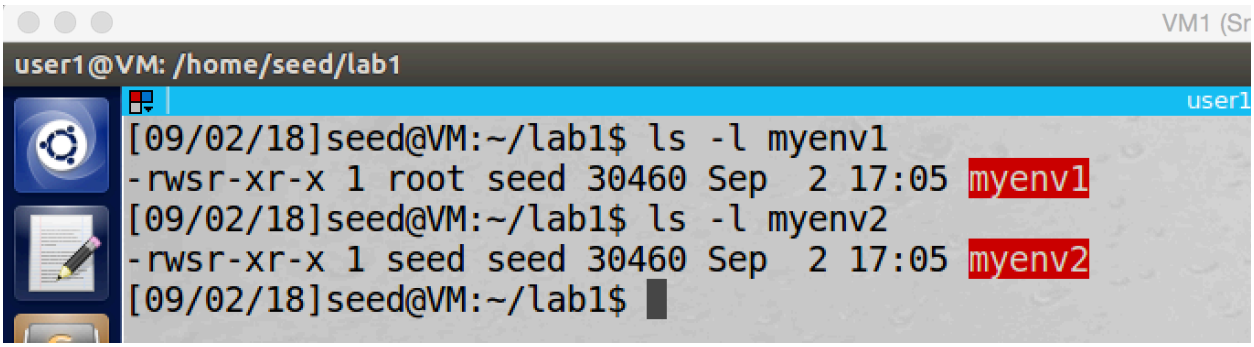
In conclusion, when the process's real user ID and effective user ID are not the same, then the LD_PRELOAD environment variable is ignored (the environment variable will not be inherited by child processes). In case 1, the user is normal, and the program is normal, so they have same real and effective user ID, then our attack is successful. In case 2, the program is Set-UID root program which means the effective user ID is root, but it is run by a normal user, so the real and effective user ID are different, then the attack fails. In case 3, the program is Set-UID root program, and it is run by the root account, so real and effective user ID are same, so the attack is successful. In case 4, we set the program to be a Set-UID user1 program. And then we export LD_PRELOAD and run the program by user2. Because the effective and real user ID are different, LD_PRELOAD environment variable is ignored, so the attack fails.

To verify the conclusion above, we do following experiment. We make two copies of env program, this program will print all environment variables of the current process. And then we set myenv1 to be a Set-UID root program, and myenv2 to be a Set-UID user program. Then we run them by a normal user account and to see if they have environment variable LD_PRELOAD. If my conclusion is correct, myenv1 should not have LD_PRELOAD (real and effective user ID are different). Myenv2 should have LD_PRELOAD which is set by us.

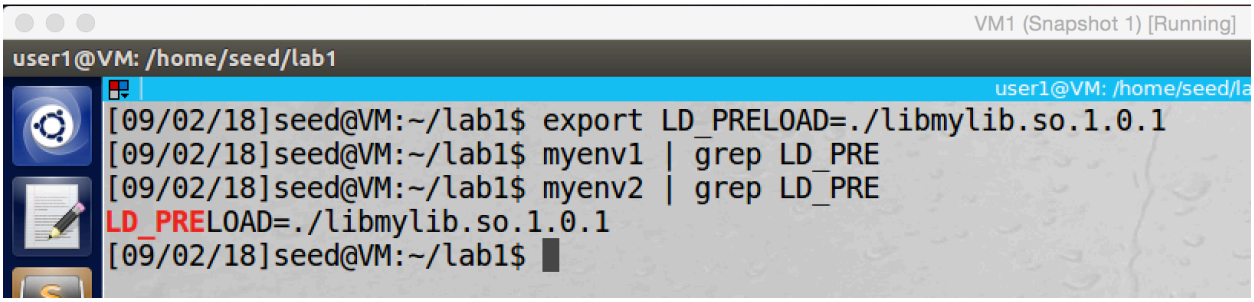


We make two copies of env program, the copies are called myenv1 and myenv2.

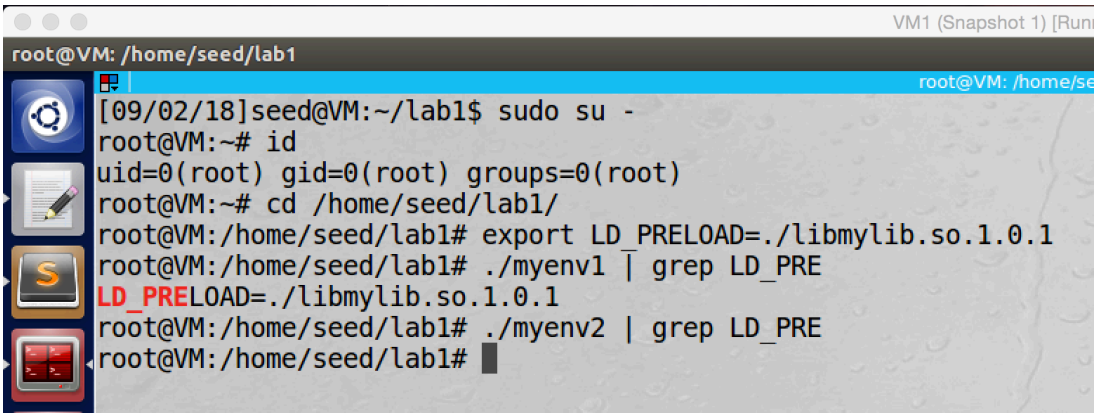




We make myenv1 to be Set-UID root program, and myenv2 to be Set-UID user program with user account seed.



And then we set LD_PRELOAD, and run these two programs by user account seed. myenv1 ignores the LD_PRELOAD, myenv2 has the LD_PRELOAD environment variable



we also switch to root account and run these two programs again. At this time myenv1 has the LD_PRELOAD environment variable, myenv2 ignores the LD_PRELOAD

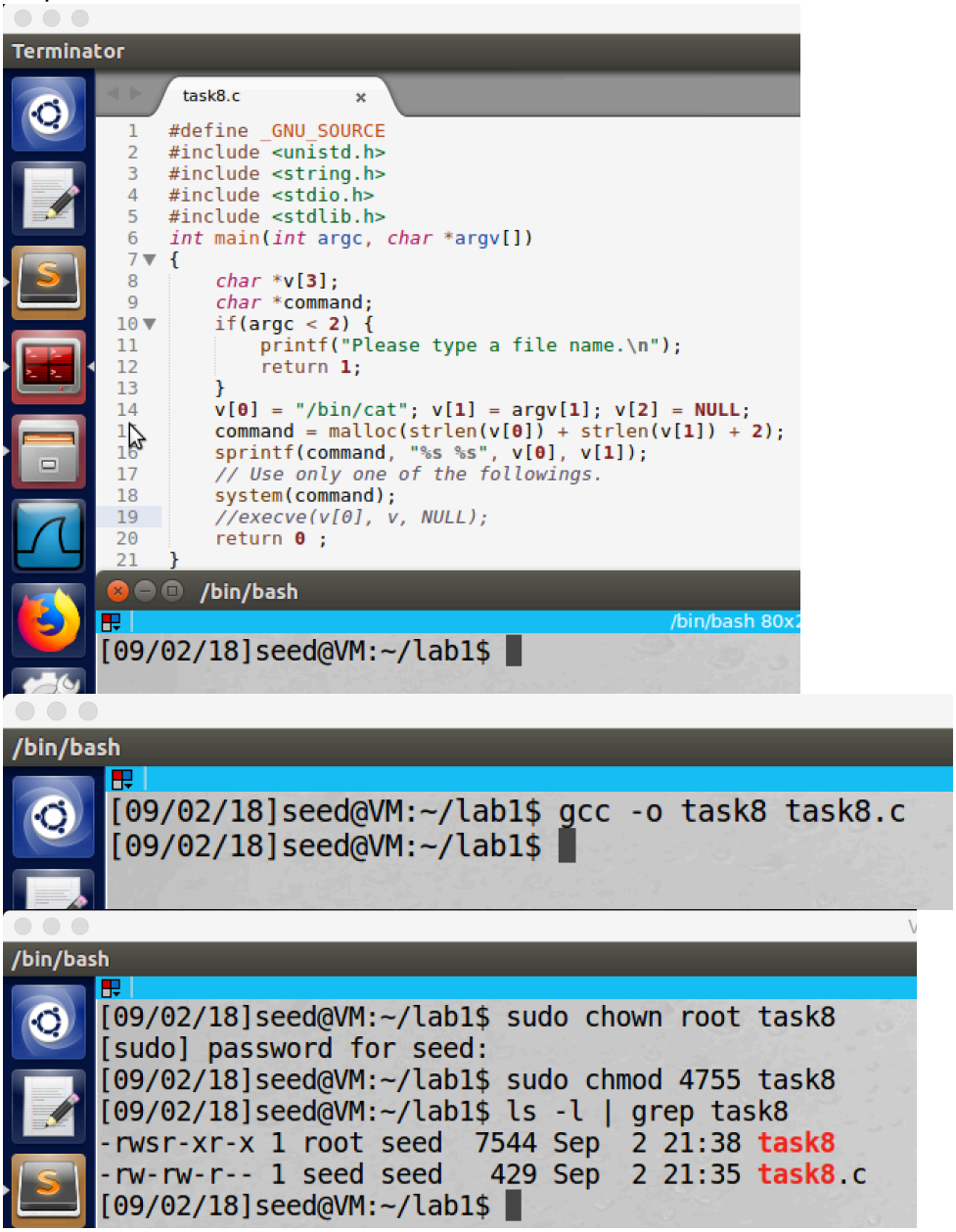
When we run these program by user account seed:
For myenv1, because its effective user ID is 0 and real user ID is 1000 (they are different), LD_PRELOAD environment variable is ignored.
For myenv2, its effective and real user ID are same (they both are 1000), so LD_PRELOAD environment variable is inherited by the child process.

When we run these programs by root account:

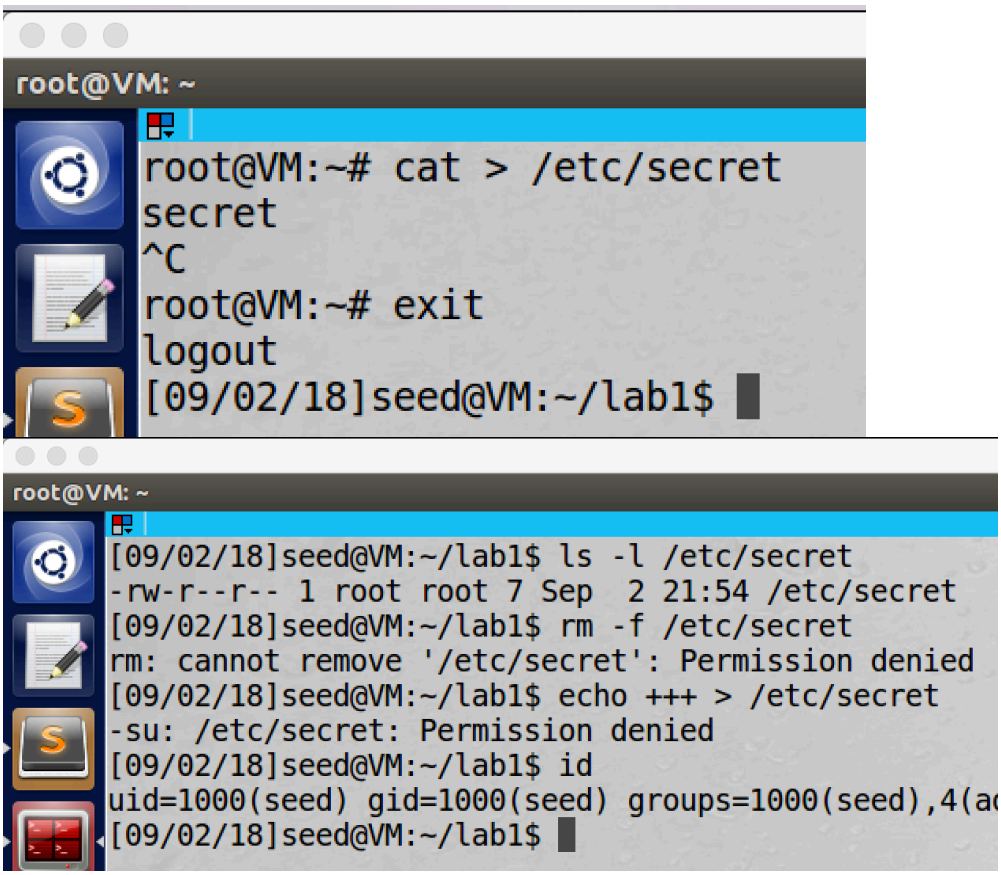
For myenv1, its effective and real user ID are same (they both are 0), so LD_PRELOAD environment variable is inherited by the child process.
For myenv2, because its effective user ID is 1000 and real user ID is 0 (they are different), so LD_PRELOAD environment variable is ignored.

The above experiment shows my conclusion is correct.

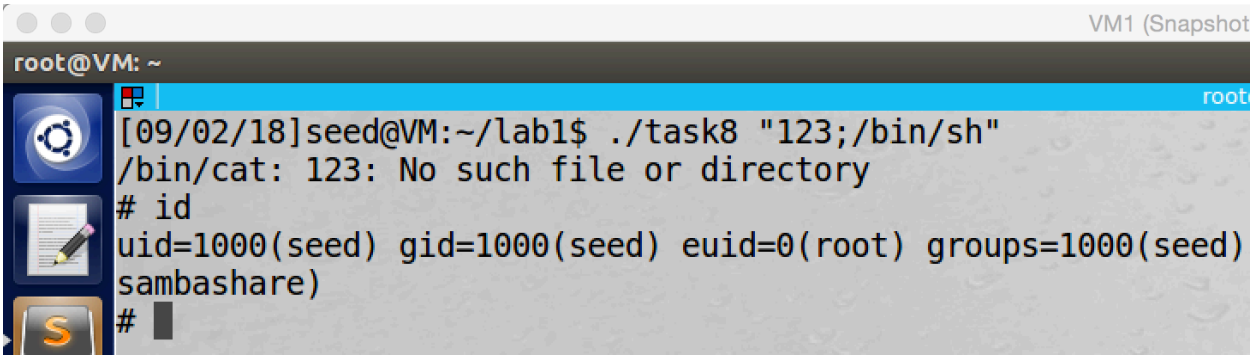
Task8: Invoking External Programs Using system() versus execve()
Step1:



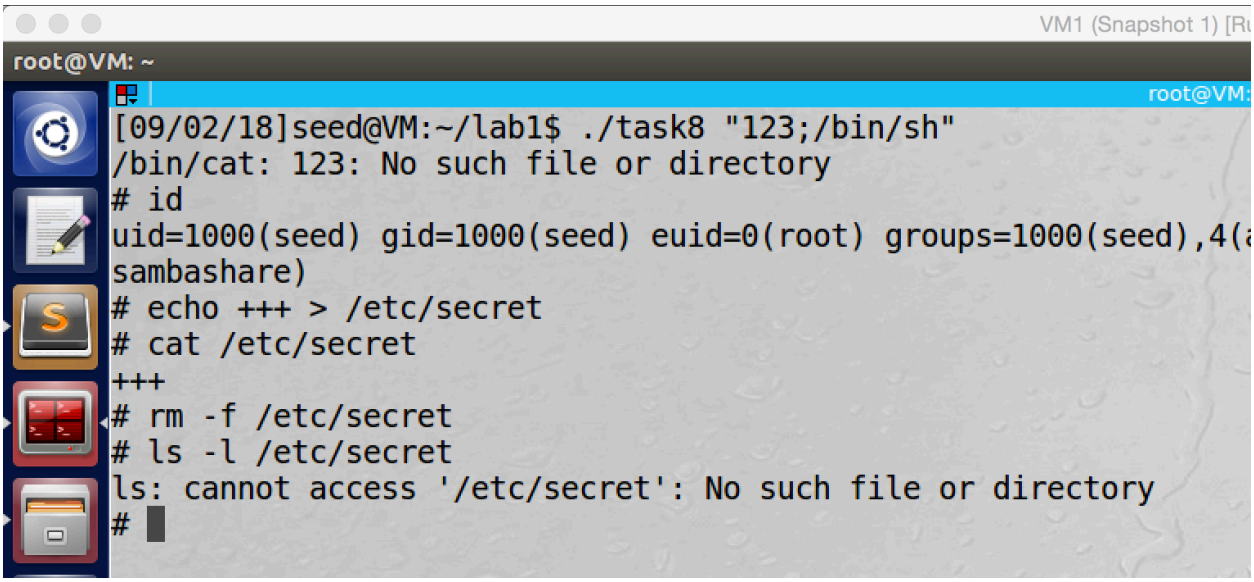
screenshot1, we copy the program from lab description and named it task8.c. And then we compile the program. Afterwards, we make the program to be a Set-UID root program



screenshot2, assuming there is a very important file which called secret (we use root account to create such file), this file is in the fold /etc. Without root privilege, we cannot remove or modify this file.

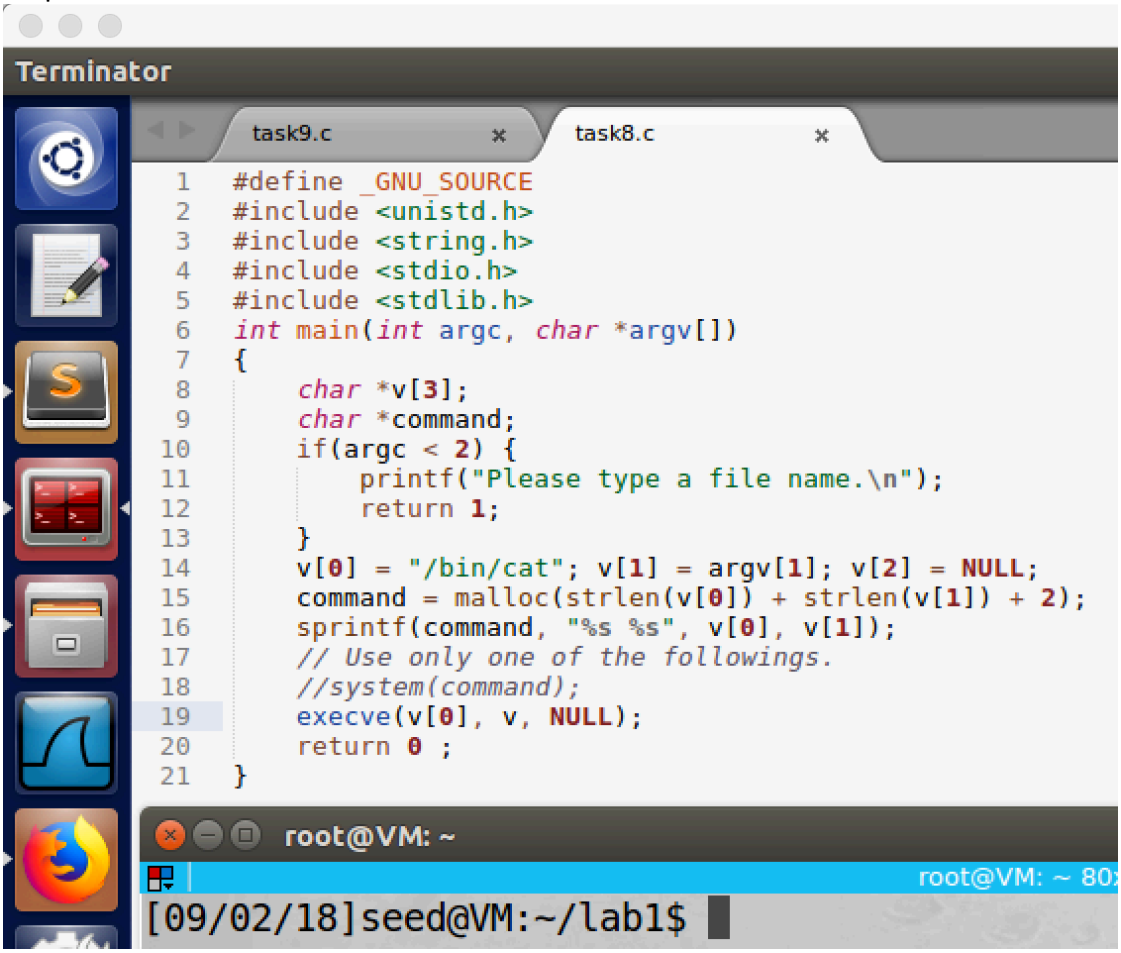


screenshot3, we run the program with input “123;/bin/sh”, then we got the root shell (eudi is 0); so now we can do anything we want, we can easily remove the secret file.

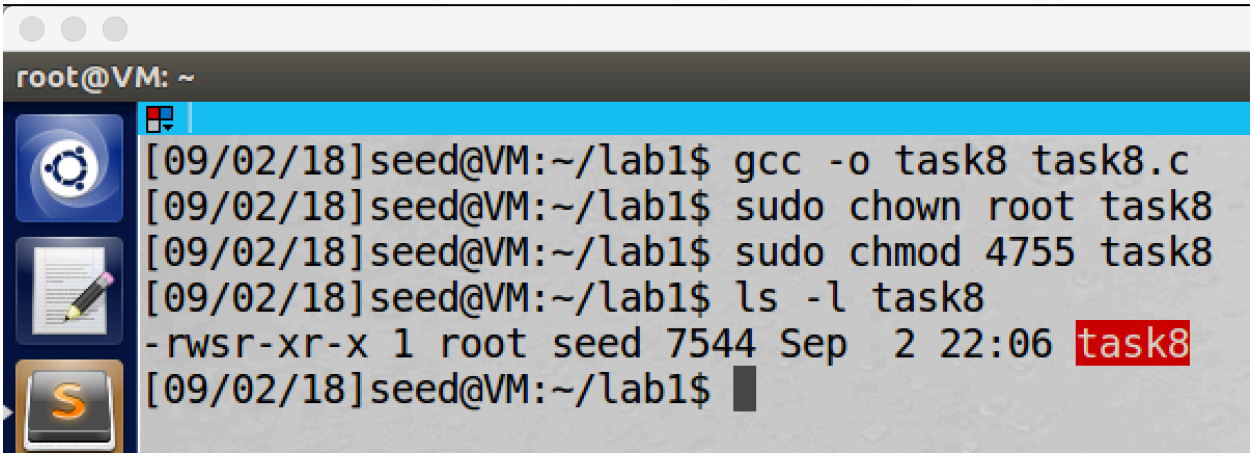


screenshot4, we use the root shell to modify and delete the secret file without any problem. So our attack is successful

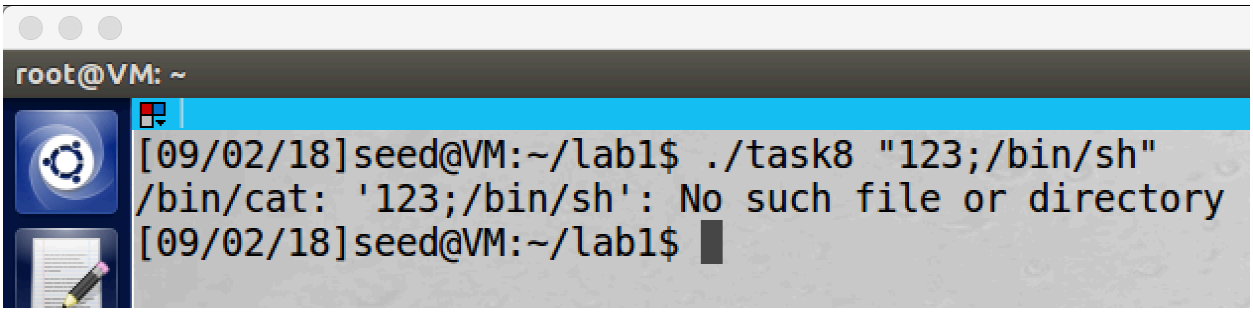
Step2:



screenshot5, this time we do not use system(); instead we use execve()



screenshot6, we compile the program, and then we set it to be a Set-UID root program



screenshot7, we try the same thing, but this time our command cannot be run. So our attack fails.

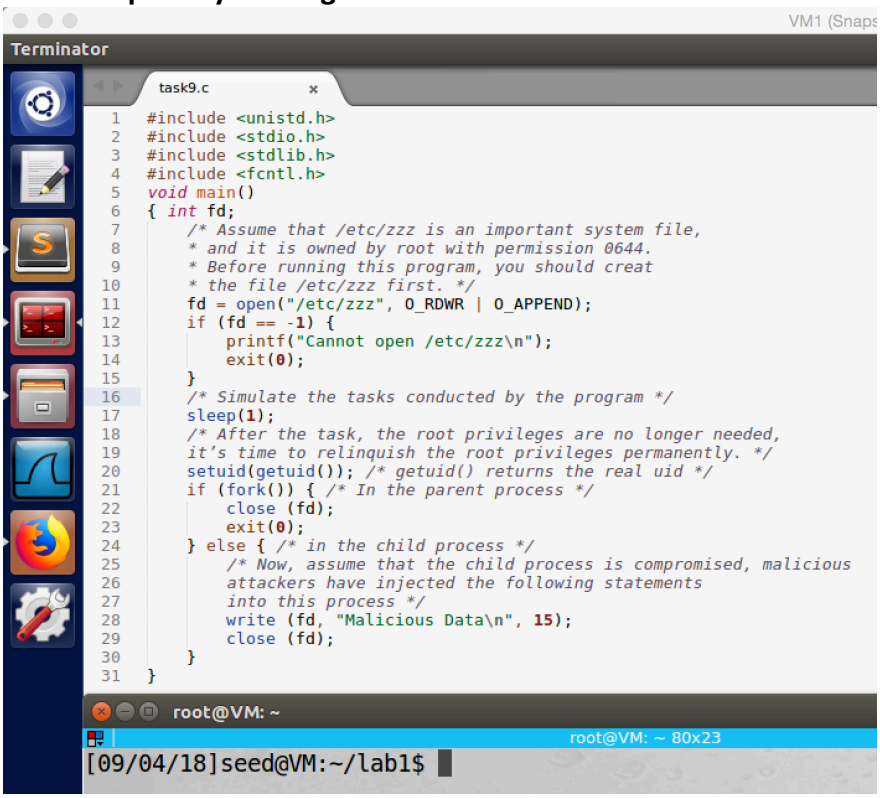
Observation and Explanation:

In this task, we are given a Set-UID root program. Such program will take a file name as input from user, and then it will open the corresponding file. In the step1, system() is involved to open the file. The program actually run command system("/bin/cat filename"). However, there is a problem, system() does not execute the external program, it actually call shell program to execute it; and the shell program can take multiple command in the same line. So the user can add additional command followed by the filename, and such command will be executed. We did same thing in step1, we first compile and make the program to be a Set-UID root program (screenshot1). And then we create a file: secret under /etc, user cannot modify and remove it without root privilege (screenshot2). Afterwards, we run the program, but the input is "123;/bin/sh"; "123" is just a fake filename, it is not important. The trick is after the semicolon, the shell program will use semicolon to separate command in the same line, so "/bin/sh" is another command. By running such command, we can get root shell. As the screenshot3 shows, we successfully to get a root shell (the effective user ID is 0). Then we can easily modify and delete the secret file. (screenshot4).

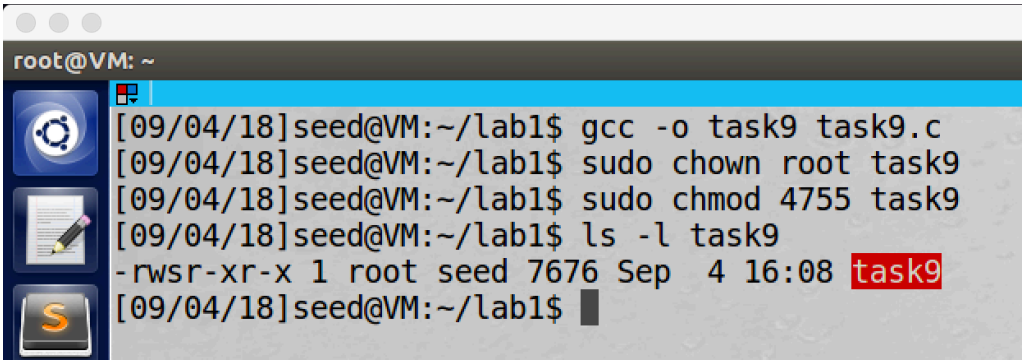
In step2, we change the program a little bit; instead of using system(), we use execve() (screenshot5). And then we compile and make the program to be Set-UID root program (screenshot6). Afterwards, we run the command again. However, this time our attack fails. The program takes the user input as whole, not separate command (screenshot7). This is because

execve(). execve() takes three arguments: the command to run, the arguments for the command, and the environment variables. And then execve() will directly ask the OS to execute the command. Therefore, the program takes our input (“123;/bin/sh”) as one single argument to the command. In other words, the additional command (“/bin/sh”) will be treated as an argument, not a command; so the attack fails.

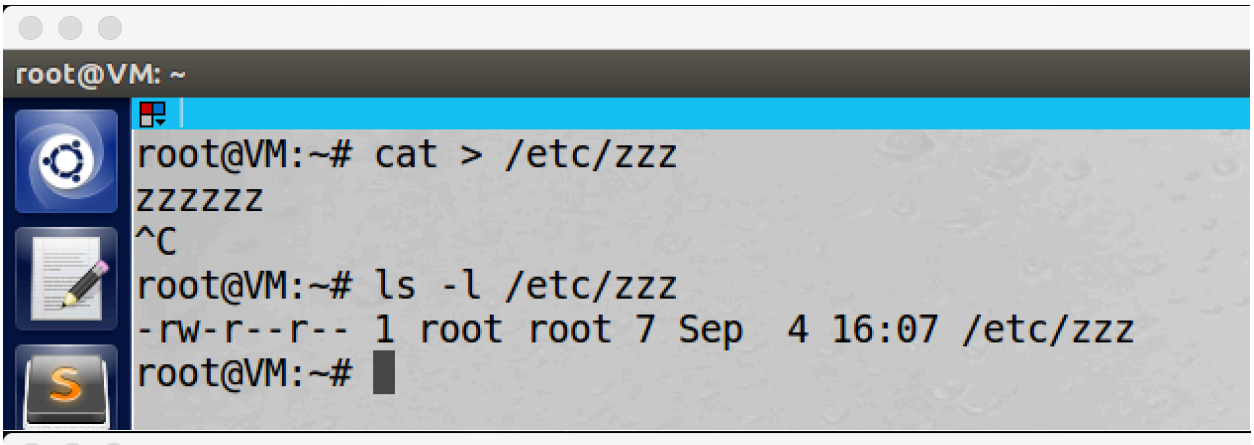
Task9: Capability Leaking

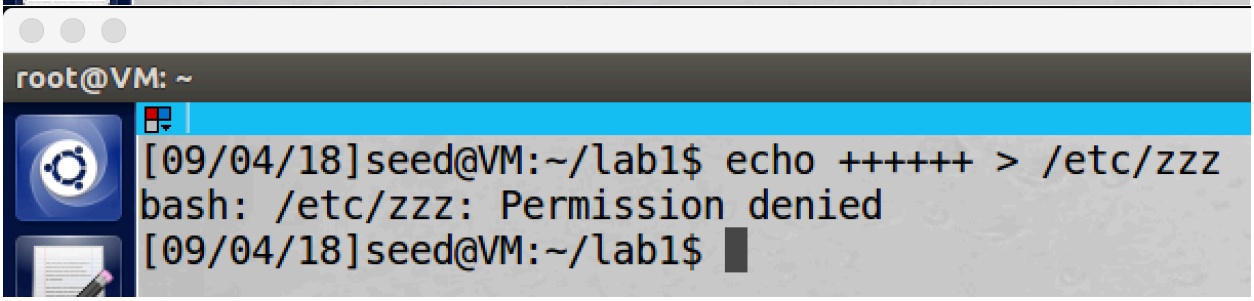


screenshot1, getting program from lab description

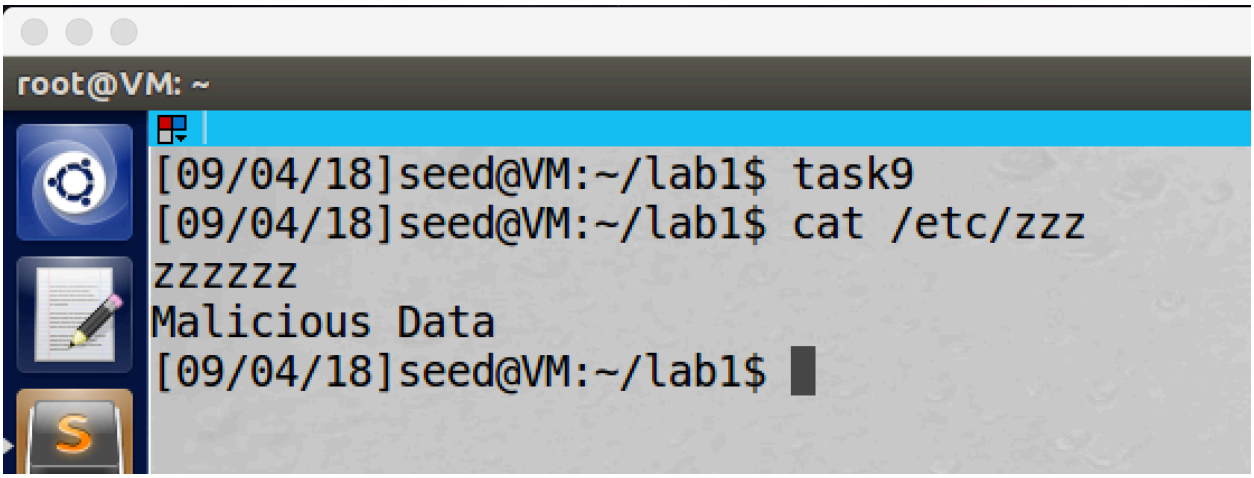


screenshot2, compiling the program, and then we make the program to be a Set-UID root program.





screenshot3, we use root account to create the zzz file under /etc. Without root privilege, we cannot modify the file



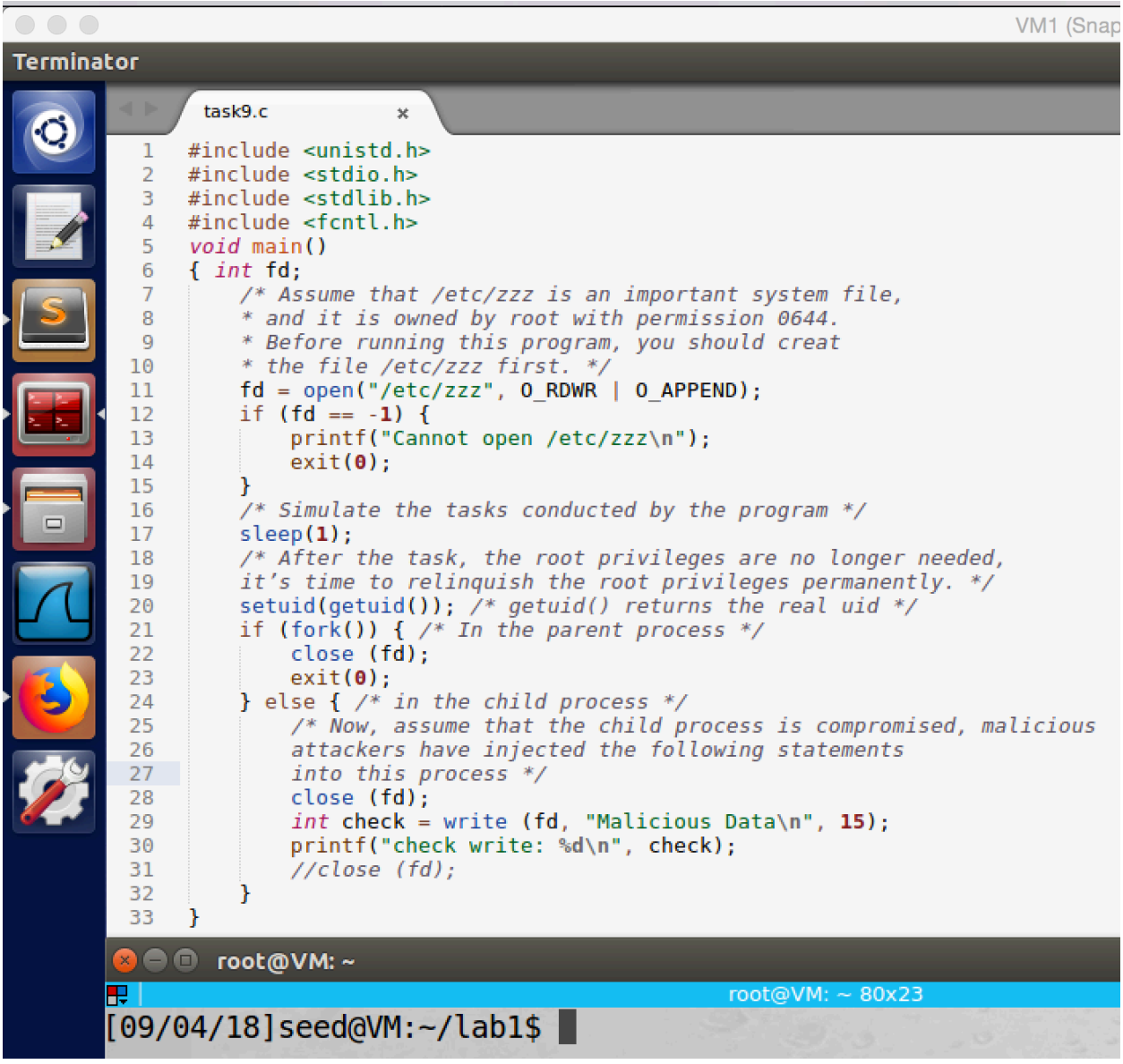
screenshot4, we run the program by normal user account, and then we check the zzz file. We see that the file has been modified.

Observation and Explanation:

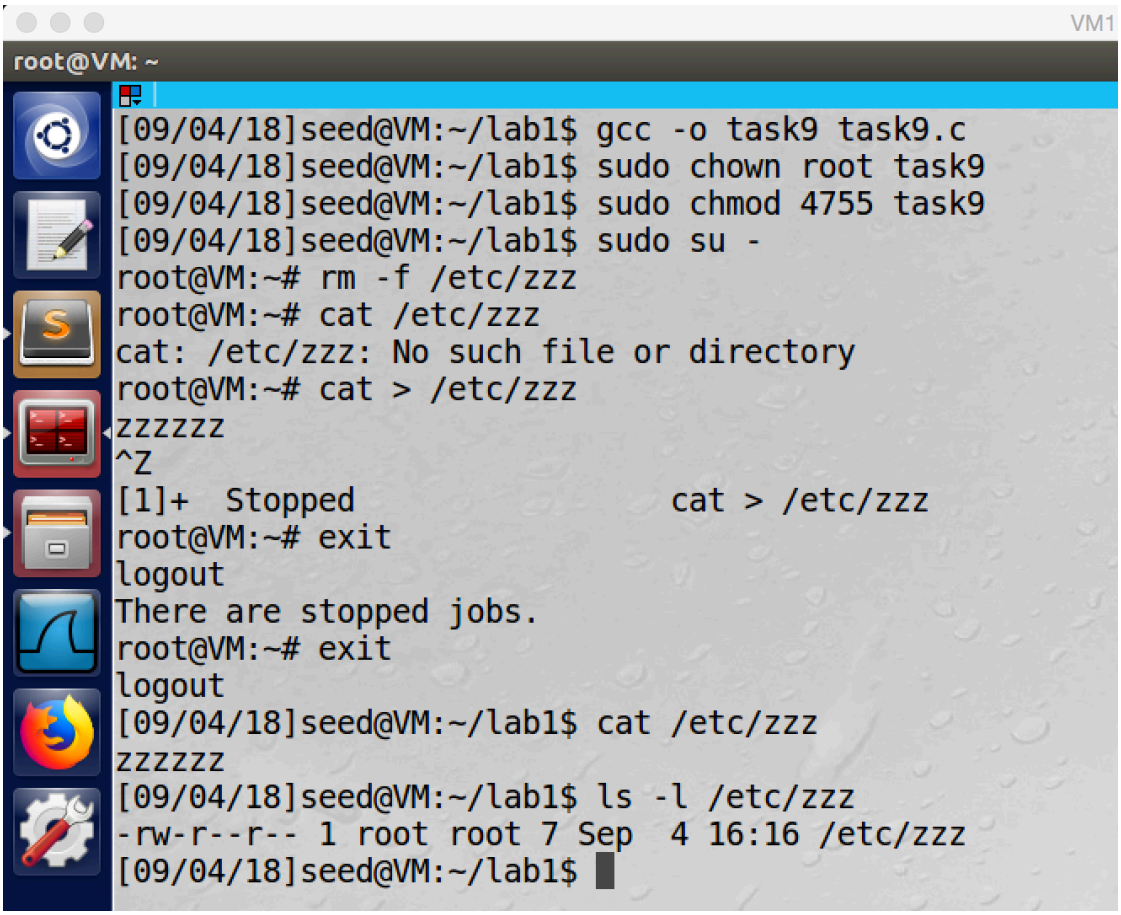
In this task, we are going to show the leaking of capability. First, we compile the program from lab description, and then we make it to be Set-UID root program (screenshot 1, 2). Afterwards, we create a file which called zzz by root account, and this file is in the folder /etc

(screenshot3). Finally, we run the program by a normal user account, and we see that the file zzz has been modified.

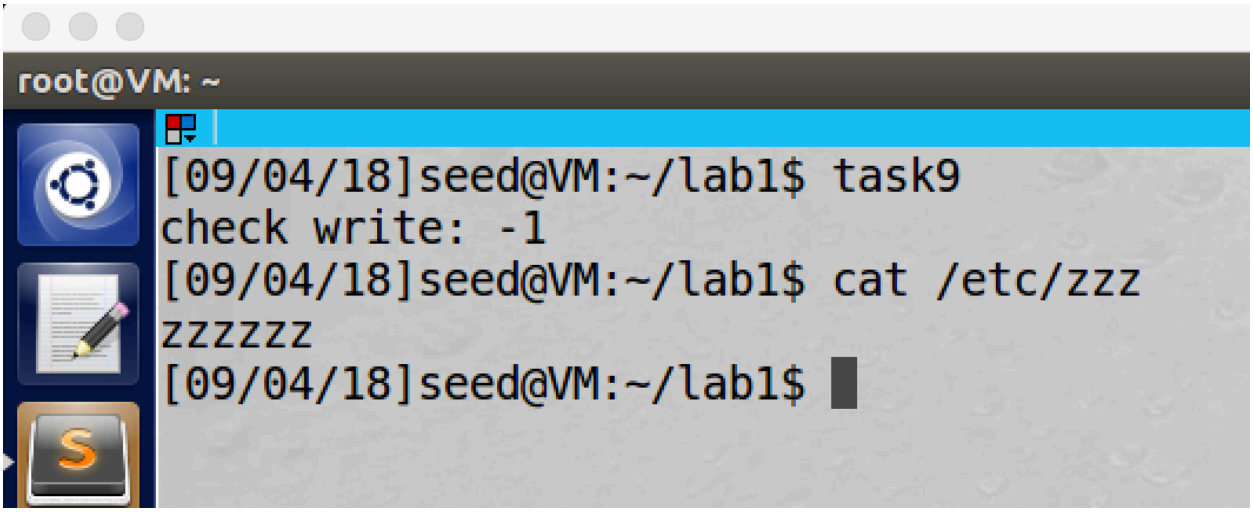
The program is a Set-UID root program, but it drop its privilege by running setuid(getuid()) in the middle, this command will set the effective user ID to the current real user ID. So the program should not be able to modify zzz after dropping the privilege. However, the program still modified the content of zzz. This is because capability leaking. When the program runs, before it dropped the privilege, it opens the file zzz, and the file descriptor is stored in the variable-fd. Therefore, even the program dropped the privilege, the file descriptor is still valid before the program close the file zzz by command close(fd). As a result, it can change the content of zzz by using the file descriptor without root privilege. If we make following change to the program, it should not be able to modify the content of zzz again.



we modified the last few line of the program, we close the file descriptor before we write to it. And we also print the return value of write(), -1 means error



We compile the program again, and then we make it to be a Set-UID root program. We also create a new zzz file and put it into /etc



After we run the program, we also check the file zzz. But this time, the file is not being modified. And we also see the return value of write() is -1 which means the write fails.

Appendix (Code for each task):

Task2:

```
//task2.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);
        }
}
```

Task3:

```
//task3.c
#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, environ);
        // execve("/usr/bin/env", argv, NULL);
        return 0 ;
}
```

Task4:

```
//task4.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
        system("/usr/bin/env");
        return 0 ;
}
```

Task5:

```
//task5.c
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
void main()
{
        int i = 0;
        while (environ[i] != NULL) {
                printf("%s\n", environ[i]);
                i++;
        }
}
```

Task6:

```
//task6.c
//#include <stdio.h>
#include <stdlib.h>
int main()
{
        system("ls");
}

//ls.c
//#include <stdio.h>
#include <stdlib.h>
int main()
{
```

```
        system("/bin/sh");
}
```

Task7:

```
//mylib.c
#include <stdio.h>
void sleep (int s)
{
        /* If this is invoked by a privileged program,
        you can do damages here! */
        printf("I am not sleeping!\n");
}

//myprog.c
#include <unistd.h>
/* myprog.c */
int main()
{
        sleep(1);
        return 0;
}
```

Task8:

```
//task8.c
#define _GNU_SOURCE
#include <unistd.h>
#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]; 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 ;
}
```

Task9:

```
//task9.c (before change)
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
void main()
{ int fd;
        /* Assume that /etc/zzz is an important system file,
        * and it is owned by root with permission 0644.
        * Before running this program, you should creat
        * the file /etc/zzz first. */
        fd = open("/etc/zzz", O_RDWR | O_APPEND);
        if (fd == -1) {
                printf("Cannot open /etc/zzz\n");
                exit(0);
        }
        /* Simulate the tasks conducted by the program */
        sleep(1);
        /* After the task, the root privileges are no longer needed,
        it's time to relinquish the root privileges permanently. */
        setuid(getuid()); /* getuid() returns the real uid */
        if (fork()) { /* In the parent process */
                close (fd);
                exit(0);
        } else { /* in the child process */
                /* Now, assume that the child process is compromised, malicious
                attackers have injected the following statements
                into this process */
                write (fd, "Malicious Data\n", 15);
                close (fd);
        }
}

//task9.c(after change)
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
void main()
{ int fd;
```

```
        /* Assume that /etc/zzz is an important system file,
        * and it is owned by root with permission 0644.
        * Before running this program, you should creat
        * the file /etc/zzz first. */
        fd = open("/etc/zzz", O_RDWR | O_APPEND);
        if (fd == -1) {
                printf("Cannot open /etc/zzz\n");
                exit(0);
        }
        /* Simulate the tasks conducted by the program */
        sleep(1);
        /* After the task, the root privileges are no longer needed,
        it's time to relinquish the root privileges permanently. */
        setuid(getuid()); /* getuid() returns the real uid */
        if (fork()) { /* In the parent process */
                close (fd);
                exit(0);
        } else { /* in the child process */
                /* Now, assume that the child process is compromised, malicious
                attackers have injected the following statements
                into this process */
                close (fd);
                int check = write (fd, "Malicious Data\n", 15);
                printf("check write: %d\n", check);
                //close (fd);
        }
}
```