ITP 30002-01 Operating System, Spring 2020

Homework 1

Name : Min Jae Yi (이민재) **Student number** : 21800511 **Email Address** : 21800511@handong.edu

1. Introduction

The main part of the hw1 can divided into two parts making the user-level application "jerry.c" and the LKM which runs on the kernel-level, and we can create the LKM "mousehold" by running the "mousehold.c".

The requirement or function for each c program can be shown as below

jerry.c
1-1. Provide API to communicate with the LKM in a user-level.
1-2. Understand the user's command and deliver to LKM in a readable format.
1-3. If username is given change as an uid (user ID).
mousehole.c
2-1. Provide API to receive information from jerry.
2-2. When module is installed change the sys_call_table as an customized function for sys_open() and sys_kill()
2-3. The module should identify who is giving command and who made the command.
2-4 Check permission for user and execute each command.
2-5 Customized opening and killing function can block command given from syscall.

Table 1 : problem define

Brief introduction for each problem.

a. 1-1 and 2-1 can be handled by defining the write function of LKM.
b. 1-3 can be handled by using the popen.
c. 2-2 can be handled by module_init() which runs after the module is initiated.
d. 2-3, 2-4 can be handled by using the structure of "current" and "task_struct" structure.
e. 2-5 can be handled by customizing the syscall function
f. 1-3 can be handled by making user application

Table 2 : problem solution

2. Approach

To explain more detail about the approach of the solution I will follow the Table 2.

a. To provide the API for user application and LKM mousehole_proc_write() which is defined for "function operation" for ".write" was used. Mousehole_proc_write is executed when the /proc/mousehold is written. /proc/mousehold/ can be written by fputs() given by c program. Delivering the string format ro the mousehole_proc_write() can receive data from the user_level. After receiving the string in the user buffer LKM will interpret the meaning of the command and give the permission for each syscall.

b. To get userid from username command "id -u username" is needed. By using the popen the linux command can be executed during the program and give the result from the file pointer. After receiving the userid the program can continue run next statement.

c. Looking at the table by kallsyms_lookup_name() we can reallocate the function for each syscall functions. By replacing the syscall function from the customized function such as "mousehole_sys_open" and "mousehole_sys_kill" and make the bridge between the original syscall function the user can check the permission before executing any syscall.

d. In the kernel-level the handler for syscall should know who is the commander, who is the creator of process and who is the current user to allocate permission for each tasks. By using the structure from "cred.h" and "sched.h" we can get information and save information in a kernel-level. Reading current->cred->uid gives information for current using user and commander. (*task_struct)->cred->uid gives the process creator's uid.

e. Blocking syscall handler function can be handled through the user defined functions which is "mousehole_sys_open" and "mousehole_sys_close". Because the user defined function gives the bridge for syscall and syscall handling functions we can block the bridge and give the wrong way to the syscall. So the bridge can examine the permission of each task and give the right job for each task. To give wrong way to the original syscall function I used sys_open(NULL, …) for file opening and sys_kill(0,sig) for killing process.

3. Evaluation

The Demo for evaluation is provided in Youtube.

https://www.youtube.com/watch?v=PlwI-KgJbpU&feature=youtu.be

	No event	*User1 blocked User2*
User1	User1 can read file User1 can change state	User1 can read file User1 can change state
User2	User2 can read file User2 can change state	User2 can't read file User2 can't change state

Table 3 : Block File Opening of User

	No event	*User1 gives user2 as user_ name*
User1	User1 can kill User1 can change state	User1 can't kill user2 User1 can change state
User2	User2 can kill User1 can change state	User2 can kill. User1 can't change state

Table 4 : Prevent Killing of Processes

To evaluate the program Table 3 and Table 4 show the working performance for each situation. To explain more about table the upper bound explains about the event and "No event" means there is no command for user of application. The left bound refers to each user and shows a perspective of each user and behavior. By testing all of the situation given in the table I confirmed that all of the tasks were well performed.

4. Discussion

One of the hardest problem I met was blocking the function of sys_open() if the permission was not satisfied. I tried many ways to solve this problem. At first I tried to use sys_close() in the function to close the file immediately. However I found that because the parameters are different with sys_open() it cannot be executed. Also looking at the library of syscall.h I found that there was sys_exit(int errorcode) syscall. However, trying to execute this function by giving sys_exit(1) I found that there was no way to execute this function. I also tried "retrun 0" and the result was never ending function execution. Finally I found that giving the wrong address of sys_open() as sys_open(NULL, …) automatically exits the sys_open(). I also struggled with the similar type of problem for sys_kill() and as I found the solution for sys_open() by giving sys_kill(0,sig) It solved the problem.

Another problem I struggled for long time was finding the User ID of the process. By looking at the task_struct source I found that task_struct->cred->uid provides the User Id for each program. I used for_each_process macro to find out the matching task_struct for the give pid. However, I couldn't find way to get UID of process by not using loop statement.

Beyond the programming of code I also had difficulty using vi and reading the header file by looking at the reference. Although I am still not familiar with these task I could learn and experience much about using these tools.

5. Conclusion

This homework shows the way of user-level application communicate with the kernel and handling the kernel-level functions. Loadable Kernel Module (LKM) provides these two methods. By initiating the module in the kernel we can touch the kernel-level through the 'write' function of module and also by changing function for syscall user can define the function of kernel.