Sand-Box

CMSC 421 - FALL 2019

Ahmed Elgendy

User-Space Version

We are required to make 4 user space programs for this project.

- Three programs for each new system call
- The fourth user-space program you must provide with your assignment is a program called "sbx421_run" to run any other application installed on your system with the credentials of a specific user, with a list of system calls from start up.

FIRST USER-SPACE PROGRAMME

Is basically a wrapper to the implemented system call “sbx421_block”

To achieve that, the code should have the following functionalities:

- Take 2 parameters with command line argument, which are {pid_t proc, unsigned long nr} the process PID and the system call number
- The program will do the following
    1. Check if the caller is root, if not, end the program with - **EACCES**
    2. Check if the pid is zero or not, if zero, then get it’s PID
    3. If not zero, check if it’s less than zero end the code with – EINVAL, otherwise continue
    4. Check the system call function number if it’s a valid number or not, if not terminate the program with – EINVAL
    5. If we reached here, this means that the passed arguments is valid and the caller is the root
    6. Call the “sbx421_block” system call using syscall() function, and pass the given parameters, and print the return 0 on success, or error massage to stderr using perror ()

SECOND USER-SPACE PROGRAMME

Is basically a wrapper to the implemented system call “sbx421_unblock”

To achieve that, the code should have the following functionalities:

- Take 2 parameters with command line argument, which are {pid_t proc, unsigned long nr} the process PID and the system call number
- The program will do the following
    1. Check if the caller is root, if not, end the program with - **EACCES**
    2. check if it’s less than one end the code with – EINVAL, otherwise continue
    3. Check the system call function number if it’s a valid number or not, if not terminate the program with – EINVAL
    4. If we reached here, this means that the passed arguments is valid and the caller is the root

5. Call the “sbx421_block” system call using syscall() function, and pass the given parameters, and print the return 0 on success, or error massage to stderr using perror ()

THIRD USER-SPACE PROGRAMME

Is basically a wrapper to the implemented system call “sbx421_count”

To achieve that, the code should have the following functionalities:

- Take 2 parameters with command line argument, which are {pid_t proc, unsigned long nr} the process PID and the system call number
- The program will do the following
    1. Check if the caller is root, if not, end the program with - **EACCES**
    2. Check if it’s less than zero end the code with – EINVAL, otherwise continue
    3. Check the system call function number if it’s a valid number or not, if not terminate the program with – EINVAL
    4. If we reached here, this means that the passed arguments is valid and the caller is the root
    5. Call the “sbx421_block” system call using syscall() function, and pass the given parameters, and print the return 0 on success, or error massage to stderr using perror ()
    6. On success print the returned number from calling the system call

FORTH USER-SPACE PROGRAMME

Is basically sand box

Description:

program called "sbx421_run" to run any other application installed on your system with the credentials of a specific user, with a list of system calls from start up. The list of system calls will be provided in a plain text file (the name of which will be given as an argument to the program) as a whitespace-separated list of numbers. As an example, if we wanted to run the command "gcc -v" as the user bob with a list of blocked system calls in the file "/etc/blocked_syscalls", we should be able to do the following from the correct directory, as root: ./sbx421_run bob /etc/blocked_syscalls gcc -v .

You need not parse any of the arguments given on the command line after the filename for the list of system calls — just run the program as specified. If the user specified does not exist on the system or the filename specified for blocked system calls cannot be opened or parsed successfully, then you should print out an error message to stderr and exit without running the command given.

Design:

1. I'll code it in C and python
2. To achieve this task, I'll need to create another code called "block_process_from_file" that takes 2 arguments {process id , file name that contains the list of process}, this is a helper program will be called from "sbx421_run" using execp() function
3. Now at the main program:
    1. Read and parse the command line argument
    2. create 2 children using fork, now as a parent I know the PID of both processes
    3. call the first process child A and the second child B
    4. in the first child process (A), call the function execvp() with the arguments of calling the program block_process_from_file to block the process before even it runs, after that is finished, then I end this process, where the parent was waiting and then run the executable that was given to the "sbx421_run"
    5. That's it

KERNEL PORTION:

Data structure to use:

As this will need multiple search entries, so the search has to be as fast as possible, and for doing that, I'll save the data for (skip-list or AVL) as I already implemented it in the last project, I'll make an array of skip lists of size [number of syscalls], that marks each system call.

The data structure will have the following main functions:

1. Int Insert(syscall number , process id)
    a. This function will insert the process id to the (skip-list or AVL) used to save the given syscall, and returns 0 on success or -1 if it was already inserted
2. Int delete(syscall number , process id)
    a. This function will remove the process id to the (skip-list or AVL) used to save the given syscall, and returns 0 on success or -1 if it was not already inserted

3. Node * find(syscall number , process id)
    a. Returns the node of the given details, or null if it was not blocked
4. Int is_bocked(syscall number, process id)
    a. Return 1 if the given process was blocked from accessing the given syscall, 0 other wise
    b. If it was blocked, then increment the counter of the number of times
5. Int count(syscall number , process id)
    a. Returns the number of times this process tried to access this syscall, or -1 if the process was not blocked
6. Void init()
    a. To initalise the data structure

FILES TO CHANGE IN THE KERNEL:

Until now I have three approaches, not sure which one to follow.

First approach:

I found out that each system call before it runs, it calls an assembly code that passes the syscall number, and the system call parameters to the registers

So I need to somehow, not sure yet, to inject an if condition there to check whether the syscall is blocked or not

Second approach:

I think according to the linux design that the assembly code mention above, leads to another C code, I can't find it yet, and not 100 % sure that it really exits, but if I found it I can easily add my if condition, that checks for blocking

Third approach:

This one is the most naïve solution, but still effective and easily achievable.

I know all the syscalls from the file "arch/x86/entry/syscalls/syscall_64.tbl" so I can look for each syscall in the kernel and inject my if condition there

These are the three approaches I am thinking about, I may find one more applicable approach.

Thank you