CMSC 421
Final Project Design

[Ahmed Almehrzi]

[Dec-5-19]

1. Introduction
   1.1. System Description

The project aims to add a new functionality to the linux kernel, which adds another layer of security to the system, this project creates a sand-box, which prevents a program from calling specific system calls, and counts the number of times this program tried o access this system call.

   1.2. Kernel Modifications

      1. Sand_box.h
         a. File that incudes all definitions of my sand_box, and all functions in it that I implemented
         b. This file is located at “include/linux”.
      2. Sand_box.c
         a. File which implements all the function referenced at sand_box.h and implements the newly added system calls
         b. This file is located at “project2/proj2/kernel”
      3. arch/x86/entry/common.c
         a. Function modified: `__visible void do_syscall_64(unsigned long nr, struct pt_regs *regs).`
            i. I added my if condition to check whether the passed system call is blocked or not, by the caller

2. Design Considerations
   2.1. System Calls and Data Structures Used

The system uses three system calls:

- **`sbx421_block`**`(pid_t proc, unsigned long nr)`
  - `which blocks the given process, from accessing “nr” system call`
- **`sbx421_unblock`**`(pid_t proc, unsigned long nr:`
  - `which unblocks the given process, from accessing “nr” system call`
- **`sbx421_count`**`(pid_t proc, unsigned long nr:`
  - `which returns the number of times, this process, tried to access this system call`

To achieve this functionality I had to think of a data structure that has very low run, time, at first I was thinking about using skip-list from previous project, but the skip-list occupies unnecessary more memory, so I went with choosing the AVL, that guarantees that all its operations is done in log(N) time, not log(N) with high probability like the skip-list.

The Main data structure is an array of AVL trees, with the same size as the number of system calls in my system, so that I can access the tree of each system call, in O(1) time.

   2.2. User-space Programs

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 program

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. Parse the command line arguments, and check if they are correct or not
  2. Check if the caller is root or not
  3. Print success message or, the handle the error number via per() method

Second User-space program

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. Parse the command line arguments, and check if they are correct or not
  2. Check if the caller is root or not
  3. Print success message or, the handle the error number via per() method

Third User-space program

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. Parse the command line arguments, and check if they are correct or not
  2. Check if the caller is root or not
  3. Print success message or, the handle the error number via per() method

Forth User-space program

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.

3. System Design

3.1. System Calls and Data Structures Used

First let's describe how the data are saved, and which data are saved:

Each system call should have it's own AVL tree, that keep track of each blocked process, and the count of that process trying to access the system call. So I created an array of the same length as the number of system calls, "`NR_syscalls`" which is a define in the kernel.

Steps done when each system call is done:

- **`sbx421_block(`**`pid_t proc, unsigned long nr)`
  1. check for root, using the current struct
  2. check for the proc if it's valid or not
  3. if it's equal to 0, then use the pid of the caller process
  4. call a function called "block_process(); that tries to insert the proc, to the nr ALV
  5. when succeeded we return 0
  6. otherwise we return error number
- **`sbx421_unblock(`**`pid_t proc, unsigned long nr)`
  1. check for root, using the current struct
  2. check for the proc if it's valid or not
  3. call a function called "unblock_process(); that tries to delete the proc, to the nr ALV
  4. when succeeded we return 0
  5. otherwise we return error number
- **`sbx421_count(`**`pid_t proc, unsigned long nr)`
  1. check for root, using the current struct
  2. check for the proc if it's valid or not
  3. call a function called "block_process(); that searches for the proc, to the nr ALV
  4. when succeeded we return count of each time that process tried to access that syscall
  5. otherwise we return error number

Files changed in the kernel code:

- arch/x86/entry/common.c
    - Function modified: `__visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)`.
        - I added my if condition to check whether the passed system call is blocked or not, by the caller
        - This function is called from the assembly code from "`arch/x86/entry/entry_64.S`" and it passes a struct that has variables for all the registers.
        - This functions is a void so it returns nothing, but it calls the system call that it's number is "nr" and it saves the return value in the "ax" register from the passed struct

```
__visible void
do_syscall_64(unsignedlongnr,
struct pt_regs *regs)
                              {
                                      struct thread_info *ti;

                                      if(is_blocked(nr , current->pid) || !likely(nr < NR_syscalls))
                                      {
                                              /*
                                                      safe exit to user space
                                              */
                                              syscall_return_slowpath(regs);
                                              regs->ax = -EACCES; // put the return value to the rax
                              register
                                              return;
                                      }
```

"regs"

- As we can see, I check for the system call if it's blocked or not using the function "is_blocked" and saves the output to the "rax" register,then end the function,and return to the assembly code again

User-space Programs

<u>We are required to make 4 user space programs for this project.</u>

- 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 program

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 program

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 program

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 program

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. Code is done in C++
2. To achieve this task, I’ll need to create a function called “`block_from_file`” that takes 2 arguments {process id , file name that contains the list of process}, this will call another function called “`vector<int> get_syscalls_from_file(string filename)`” that takes a file name and returns a vector that contains all the system call in that file, or terminate the program if any of the syscalls in that file is invalid.
3. Read and parse the command line argument
4. Check if the given user is a valid user or not
5. Call the function “`block_from_file`” and give it the file name and the caller process PID using getpid(), and block the caller process from the given syscalls
6. Call the passed program using execvp() function
7. That’s it.

4. References

[1] The Linux programming interface_ a Linux and UNIX system programming handbook-No Starch Press (2010)

[2] Linux insides, 0xAX, link