CMSC 421
Final Project Design

Dylan McQuaid
December 5, 2019

1. Introduction

1.1.System Description

This project creates a new version of the Linux kernel that adds functionality to support a simple permission system for using system calls. This system will operate by maintaining a list of process IDs which are blocked from using a specific system call. If a process ID is in a system calls list, it is never allowed to call that system call unless it is removed from the list.

1.2.Kernel Modifications

Modified:

- arch/x86/entry/common.c
  - do_syscall_64 was the only function modified inside this file. It was modified to include code that checks the structure implemented to see if the process attempting to do the system call has permission to do so (i.e. is not blocked)
- arch/x86/entry/syscalls/syscall_64.tbl
  - Added the 4 new system calls to the system call table
- Include/linux/syscalls.h
  - Added the prototypes of the 4 new system calls
- Makefile
  - Added the path to the kernel makefile to the makefile of the new system calls

Added:

- include/linux/proj2.h
  - header file for the new data structure added
- proj2/kernel/proj2code.c
  - File that contains the definitions of the new system calls
- proj2/kernel/Makefile
  - Makefile to build the kernel space code
- proj2/Makefile
  - Refers to the kernel/ directory Makefile

2. Design Considerations

2.1.System Calls and Data Structures Used

The system calls for this system make use of an array of skip list. A skip list is a modified, sorted linked list. As more of an explanation, it is a linked list where each node is assigned a random level of forward/next pointers. Being that each node will not have the same number of forward pointers it is possible when accessing the skip list that some nodes are able to be skipped over. As an example, if we have 10, 20, 30, 40, 50 and 60 implemented in our skip list and say all are on level 1, but 10, 30 and 50. Well a search starts at the highest level and works down, so for 50 the search will start at level 2 and be able to go across this level 10->30->50 skipping 20 and 40. If we were searching for 20 which is on level 1, the search again would start on level 2, start at 10, jump to 30 and

realize it has passed the node it is looking for so it will drop down a level at node 10 and look forward again and hit the desired node, 20. Given that system calls are made quite often in the kernel the search for whether the process calling that system call had permission to do so had to be as efficient as possible. To achieve this efficiency an array of skip list was an useful solution because with each index in the array representing the system call number, the time complexity of step 1 of the search is O(1), leaving only the search through the skip list which takes O(log n) time in the average case.

2.2.User-space Programs

The first three user-space functions, sbx421_block, sbx421_unblock and sbx421_count act as simple wrappers for the system calls. They simply perform the system calls with the arguments passed in from the command line. They also perform basic error checking to make sure the arguments passed in are valid before finally making the system call. The fourth user-space program, sbx421_shutdown acts similarly to these three, but it does not accept inputs. It is just a wrapper for the sbx421_shutdown system call.

The fifth user-space function, sbx421_run is different from the first four. This user-space program reads in a list of system call numbers from a file to be blocked, changes the user ID to the user passed in on the command line and runs a program specified on the command line. This user-space program also does some basic error checking of the values passed in

3. System Design

3.1.System Calls and Data Structures Used

As mentioned in the above “Design Considerations” this implementation uses an array of skip list. To do this, inside the proj2.h file a struct was defined for a skip list node and the skip list itself. The node containing its ID, count and forward array. The skip list containing its current level, max level, probability of adding another level and a node pointer to the header of the skip list. Some constants were defined as well, the number of system calls, the probability of levels to be added, the max level, and the max ID. The number of system calls will be used to create the size of the array for this structure. A probability of 2 was chosen to give a higher chance of multiple levels being added for a node. A max level of 31 was chosen, as according to William Pugh’s paper using a max level of k, where the skip list can contain up to 2^k elements is appropriate (Pugh). 2^31 because PIDs are int values that can only be positive so they will go up to INT_MAX. The last thing declared in the header file is an accessor function for the array.

Now into the file where the system calls themselves are defined. The first function to note is a helper function that calculates the number of levels that a node will be when it is added to the skip list of a system call. This function makes use of the get_random_bytes() function to get a random value for the level. Next is the declaration of the array itself. The array is an array of pointers to skip list and must be the size of the [number of system calls + 1]. Next is the definition of the accessor function for the array which has return of type skiplist** and simply returns the array. The next helper function is an initialization

function. This function allocates the skip list of the system call passed in if it has not been allocated already. It allocates the skip list, sets its max level, level probability, current level, allocates the header node, sets the process ID of that header node to the max ID and allocates the header's forward pointers. Finally, we loop over the size of that forward array and point them all to the header.

Onto the system calls. This first system call is the block system call. The block system call (like all the system calls in this system, other then shutdown) takes in a pid_t and unsigned long as the system call number to block. First is some error checking. There's some basic error checking such as making sure the pid_t passed in is greater than or equal to 0 and less than its max, and the same is done with the system call number passed in, if not error out with -EINVAL. In addition to those basic error checks, there is some special checks that must be done. In order for the calling process to block itself a pid_t of 0 must be passed in, so we must check for if pid_t equals 0 and set the pid_t to the current->pid if it is. Now if a process attempts to block another process from calling a specific system call the user must be root, so if pid_t is greater than 0 and the current user is not root then error out with -EACCES. Finally, the last check is to see if we need to call the initialization function described earlier. This would be the case if the skip list at array[system call #] is NULL. After the initial checks comes the skip list insert code. An update array is kept so that when the search is complete and we are ready to insert, the skip list does not lose its structure, and everything can be updated to the proper places. A node pointer x is set to the header of the skip list of the system call. We then loop over the skip list looking for the proper spot to insert the new process to be blocked. How do you loop over a skip list? Well using for loop with a nested while loop, we do the following;

```
for(i = array[system call #]->currentlevel; i >= 1; i--){
        while(x->forward[i]->PID < passed in PID){
                x = x->forward[i];
        }
        update_array[i] = x;
}
x = x->forward[1];
```

After we've found our desired location we check if the PID of the node we have decided to insert at is equal to the PID we are going to insert, if so, we error out with -EEXIST. If the PID is not a duplicate we get the random level to be assigned, update the current level of the skip list if needed and then create the new node.

The next system call is unblock. Unblock follows the same structure as block but does not have the special case involving a PID of 0. The user must be root to call the unblock system call, the PID must be greater than 0 and if the passed in system call's skip list is not allocated, we just error out with -ENOENT. The skip list code follows a similar structure for unblock as well, we have an update array again and loop over the skip list in the same fashion. Except this time if we find a matching PID to the one passed in we perform the deletion of that PID from the skip list. We perform a loop over the skip list going from i = 0 to i = current level of the list and set update[i]->forward[i] = x->forward[i]. We then free

all the allocated memory for that node and set its count to 0 (count will be mentioned in the next paragraph when talking about the count system call). Lastly, we take down the current level of the skip list if we need to. If we happen to not find the PID, meaning that process is not blocked from making that system call, we return an error of -ENOENT.

Second to last is the count system call, which is the simpler out of the bunch in my opinion. It performs the same error checking as unblock, then loops over the skip list and if it finds a node with the PID value passed in it simply returns the count variable of that node. The count variable is tied to each node, which means it is tied to each PID and system call pair. This count variable keeps track of how many times a blocked PID attempts to call a system call it is blocked from. Again, similarly, to unblock if the PID is not found we return -ENOENT.

The last system call is shutdown. This system call requires root to be called and loops over the entire array structure and checks for any system calls that currently have skip list. If there is a skip list we loop over freeing x using a temp variable until our node x, which we set equal to array[i]->header->forward[1], is equal to the header of the skip list. Once it is, we free header->forward, header and the entire skip list itself. This function was primarily used for testing but can be a quick way to reset the structure without having to reset the kernel.

With all of these system calls implemented and the data structure setup, the last step to the implementation was editing the location inside the kernel that handles making a system call. After some research the function to edit was found inside the arch/x86/entry directory inside the common.c file. The function to edit was do_syscall_64. Inside this function I made use of the accessor function to get the array and access it at the proper system call number that is passed into this function. In order to prevent the code from advancing if the PID is found inside the system call's list, I created a variable that I used as a flag as to whether the PID is blocked. If it is blocked the flag will be set to 1 and the actual code that does the system call will not be executed, otherwise, if it is 0 the code will be executed. If the PID is found to be blocked from making that system call, we set regs->ax to -EPERM so that this error is returned from the system call attempt.

3.2.User-space Programs

As referenced in the above user-space section there are a total of five user space programs. The sbx421_shutdown program just simply calls the sbx421_shutdown system call when ran. The three out of the four remaining, sbx421_count, sbx421_unblock and sbx421_block are very similar. They do error checking such as, checking if the user is root, if enough arguments (argc) were passed, and uses the strtol function to determine if the PID and system call number passed in on the command line are actually numbers and then it checks if those two numbers are between their bounds. Then they call the respective system call with the two command line arguments and prints out whether it fails or succeeds. The one difference worth mentioning, is that

when the user calls the sbx421_count program, if the system call succeeds it will also print out the count to the user.

The last user-space program, sbx421_run does some similar error checking, whether the user is root and if enough arguments are passed in. It then, however, reads in a file using the fscanf function and puts the system call numbers read in from this file into an array. While reading this file the program also does error checking to make sure each value read in from the file is a number. After the values have been read in from the file successfully, we check if the user id passed in on the command line exist using the getpwnnam() function. If the user does exist, we loop through the array where all the values were added and block the program from calling each of those system calls. We then, set the user id of the program to that of the user id passed in on the command line. Lastly we use a combination of fork() and exec() to execute to program passed in as the last command line argument. If the program fails at any of the error checks we call exit(EXIT_FAILURE) to stop execution of the rest of the program.

4. References

Sasikala. “131 Linux Error Codes for C Programming Language Using Errno.” *The Geek Stuff*, 18 Oct. 2010, www.thegeekstuff.com/2010/10/linux-error-codes/.

Pugh, William. “Skip Lists: A Probabilistic Alternative to Balanced Trees.” *Algorithms and Data Structures*, Edited by Jeffrey Vitter, vol. 33, no. 6, June 1990, pp. 668–676.

Alex. “How the Linux Kernel Handles a System Call.” *System Calls in the Linux Kernel*, https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-2.html.