CMSC 421
Final Project Design

Andrew Ingson
December 6th, 2019

1. Introduction

1.1. System Description

In sumuray, the goal of this project was to create a simple permission system for processes using syscalls by implementing ‘sandboxing’. Sandboxing simply refers to the practice of creating a sequestered environment for each process to ensure it has the resources it needs, and only just those resources. For this project, sandboxing is used block/limit which syscalls a process can run.

1.2. Kernel Modifications

1.2.1. `Makefile` - Modify the makefile to have a relevant version string (line 5) and to include the project 2 source code folder (line 996).

1.2.2. `arch/x86/entry/syscalls/syscall_64.tbl` - At line 358, add the table entries of the three syscalls, sbx421_block, sbx421_unblock, sbx421_count. All calls should be common

1.2.3. `include/linux/syscalls.h` - At line 1400, add the three asmlinkages for the previous three syscalls. All three syscalls should return *long*. All three should pass the parameters *pid_t proc* and *unsigned long nr*.

1.2.4. `arch/x86/entry/common.c` - Modify line 287 to have an if statement. This statement should check to see if the current pid with a given syscall id is in the access control list.

2. Design Considerations

2.1. System Calls and Data Structures Used

At a high level, I decided to use an array of skip lists as my data structure for this project. For this implementation, there is a skip list of blocked processes for each syscall. Even though a skip list may not be the most efficient choice of a data structure for this use case, it was chosen primarily due to my familiarity with its implementation from project 1. Additionally, a skip list was chosen for this because my code from project 1 could be reused, therefore significantly speeding up the development process (and making my life much easier).

From a technical standpoint, skip lists have the benefit of on

average having O(log(n)) time for search, create, and destroy. Additionally, skip lists dynamically allocate memory as it used (instead of having to allocate everything at once) making it much more efficient.

The 3 required syscalls, sbx421_block, sbx421_unblock, sbx421_count are all implemented using skip lists. The block syscall essentially acts as a wrapper for the create method. The unblock syscall acts as a wrapper for the destroy method. The count syscall acts as a wrapper for the search method.

2.2. User-space programs

2.2.1. 3 simple wrapper programs (sbx421_block, sbx421_unblock, sbx421_count)

For each of the required syscalls, a simple wrapper program was developed to take arguments for the syscall and pass them to their respective syscalls. These programs take the 2nd and 3rd arguments from argv, convert them to longs, and then pass them to the appropriate syscall.

2.2.2. Sb421_run program

I was unable to make this test work properly but the goal of it was to read processes to be blocked from a file then push it into the block syscall

3. System Design

3.1. System Calls and Data Structures Used

To begin the implementation of the kernel, the libraries kernel, syscalls, zconf, string, errno, list, rwlock, time, cred, and init should all be included so we can include all the features we need.

First we need to create the structs to our data structures will use. For the syscall array struct, there should be pointers to a head and tail skip list node, a counter for how many processes are being blocked, a bool for whether the skip list is started, and int to keep track of size, and a rwlock. For the skip list node struct, there should be a pid var, an unsigned int for the height, an int for keeping track of how many times a process has

been blocked, and an array of pointer to the next node in the skip list.

Next we must set some global vars. These should be a var for the max skip list size, the probability of a skip list adding a new level, the amount of syscalls in the system, a bool for whether the sandbox system is ready, a static int for randomization, and an array of pointers to the syscall nodes.

The first method is the initialization of the sys call array. This method should set up the array so it is able to be used properly later. After we check the initialization status global, we then make a for loop to traverse each entry in the array and set it to NULL. In this way we are able to determine whether a skip list has been set up at this entry.

The next method is the skip list insert method. This method takes in an unsigned long syscall number and a pid. It first calls the init function in order to make sure everything is ready to rumble. Next we check to make sure all our arguments are sanitary and in range. If not an error code from errno.h is thrown. Next we determine if the skip list for the given syscall id has been initialized. If not, we do it. This involves making a writer lock, allocating memory to each of the pointer variables, and setting all of the other vars to there base values. For these, the pid should be -1, the height should be the max size of the skip list, the block count should be 0, the process count should be 0, the current size should be 0, and the initialization state should be set to true. After this is complete, the writer lock is released.

Once the skip list has been initialized, a read lock is made. In order to traverse the list, we need to make variables for our current traversal level, a pointer to our current node, and an array to keep track of all the nodes we have visited on the way down to our current location. We then use a for loop to move through each level and check if first at the bottom, add the current node to the history, and a while loop to move right if the right node is not the tail and it is greater than our pid. When these loops complete, we should be at our destination. At this point we check if the node already exists. If it does we unlock and throw an error. Otherwise, we unlock the read lock.

To add to the list, we must first write lock. To add a new node, we have to figure out how tall it should be. To do this, we need to loop until our probability says we should not go up or we are at the ceiling. Once this is done we can allocate space for our new node. The id for this new node is the pid arg and the height is the new height we calculated previously. Now we need to allocate space for this nodes array of pointers to the node

after it. Once this is done, we use our history array from earlier to splice in this new node in between it's new neighbors. This can be accomplished in a loop for every level up to the new node's height. After this is done, we check to see if our new height is greater than our current height. If it is, we update it as such. Finally, we free our dynamically allocated history array and unlock.

The next method is our destroy/deletion function. This method takes in an unsigned long syscall number and a pid. This process is very similar to the insert method. First, we check to make sure all our arguments are sanitary and in range. If not an error code from errno.h is thrown. Next we determine if the skip list for the given syscall id has been initialized. If not, we do it. This involves making a writer lock, allocating memory to each of the pointer variables, and setting all of the other vars to there base values. For these, the pid should be -1, the height should be the max size of the skip list, the block count should be 0, the process count should be 0, the current size should be 0, and the initialization state should be set to true. After this is complete, the writer lock is released.

Once the skip list has been initialized, a read lock is made. In order to traverse the list, we need to make variables for our current traversal level, a pointer to our current node, and an array to keep track of all the nodes we have visited on the way down to our current location. We then use a for loop to move through each level and check if first at the bottom, add the current node to the history, and a while loop to move right if the right node is not the tail and it is greater than our pid. When these loops complete, we should be at our destination. At this point we check if the node already exists. If it doesn't we unlock and throw an error. Otherwise, we unlock the read lock.

Next we make a write lock since we are removing from the list. For the node we intend to delete, we go through each of the dynamically allocated objects and free all of the memory. Once that is done, we can use the history array to stitch the skip list back together.

Following the destroy method there is the search method. This method is functionally the same as destroy except it does not include the deletion step. Once it checks to see if a node exists it returns that status.

Additionally, there is the block count increment method. This method is the same as the search method except that when it find the node it increments the block count.

To help with debugging, a print method can also be made. This method takes in an unsigned long syscall number and a pid. This process

is very similar to the insert method. First, we check to make sure all our arguments are sanitary and in range. If not an error code from errno.h is thrown. Next we determine if the skip list for the given syscall id has been initialized. If all these conditions are passed, we then increment through the next node pointers of each node in a level, printing the value of each.

Next there is the sbx421_block syscall. This call begins by collecting the current uid and pid using current. If the process id is greater than 0 and the user is root, the skip list insert method is called. If the process id is 0, the insert method is called with the current pid instead of 0. If none of these conditions are met we return an error.

Following this there is the sbx421_unblock syscall. This call begins by collecting the current uid. If the user is root (uid = 0), we run the skip list destroy call with the given values.

Finally there is the sbx421_count syscall. This call begins by collecting the uid. If we are root, we run a search the same way as above except when it finds the process it returns the value of block count.

3.2. User-space Programs

No real kernel development is done with these programs besides the fact they use syscalls defined in kernel space.

4. References

4.1. BIG O. “Know Thy Complexities!” *Big*, Sept. 2011, https://www.bigocheatsheet.com/.

4.2. Linux Community. “ERRNO(3).” *Errno(3) - Linux Manual Page*, 19 Nov. 2019, http://man7.org/linux/man-pages/man3/errno.3.html.

4.3. Ingson, Andrew P. “ADMARII - Project 1.” *GitHub*, 1 Nov. 2019, https://github.com/ADMARII.

4.4. FNAL. *Errors: Linux System Errors*, 2019, https://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Errors/unix_system_errors.html.