CMSC 421
Final Project Design

Daniel Roh
12/06/2019

1. Introduction
    1.1. System Description

A new creation of a new linux kernel that adds the feature of adding permissions to users access to individual system calls within the linux kernel. This will allow a basic sandbox in which users who use this feature can limit what can be done on the operating system.

    1.2. Kernel Modifications

- Proj2 (folder)
    - kernel (folder)
        - proj2.c
- arch (folder)
    - x86 (folder)
        - entry (folder)
            - syscalls (folder)
                - syscall_64.tbl
            - common.c
- include (folder)
    - linux (folder)
        - proj2.h

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

Given that the requirements called for the ability for any of the 436 system calls to be blocked individually for any given user. This would mean potentially a large data structure for each user. Instead, given that the system calls are a fixed number. An array with 436 slots could be made and and insure a constant access time by linking each slot in the array with the system call number. Within each slot a hashtable will be used to keep track of each user that has been blocked from using that system call. This would be because of the fact that there can be varying number of users that could be added or removed from the block list. For the hashtable, a good key would be needed to insure quick operation thus the users uid can be used as it will be unique for each user and thus insure that would be little to no collisions in the table. An array and hash table was selected as access time would be a priority in the kernel as a slow data structure would lead to a slow experience for the user in the operating system.

    2.2. User-space Programs

While the project deals with kernel space programs, to test if these kernel features work. A simple program can be created to test if the feature is working correctly. Given that there are 3 new system calls that are being implemented, it would make sense to test each one separately. The first test would be testing the blocking system call as it will be hard to test the other system calls if the block call does not work. The second test would test if a blocked call can be unblocked properly after being block initially. Lastly the third call would test if count of the amount of times a users called a blocked syscall returns the proper data.

Another Testing program sbx421_run can also be created to allow more flexible testing across more users. This program should be able to take arguments to quickly test commands as different users and blocking multiple syscall ids.

1. System Design
    1.1. System Calls and Data Structures Used

To implement simple blocking in the kernel, only two main system calls would need to be added to the kernel (a third system call can also be added (which is used in this project mentioned below). These system calls would be “block” and “unblock”. Block and unblock should be able allow a root user to block or unblock any user from using syscall in the kernel, even the root user themself. Meanwhile a normal user should only be allowed to block/unblock at most themselves from using a system call. In the case a normal user tries to block or unblock another user an error should be returned. Block and unblock will need the information of uid that will be blocked and the syscall to block for the given uid. These new system calls should return an error in the case a block/unblock failed and ideally print a message on why it failed. Also, for this project another system call named “count” would be added to check the amount of times a user tried to access a blocked system call. This system call should only be allowed to be called by the root user as it can be used for an internal security audit to see if normal users are potentially trying to make changes to kernel when blocked.

To effectively block the system calls, a single entry point would need to be found in which all of the system calls that a user requests would move the user from userspace to kernel space. A spot before the entry to kernel space to check if the user has permission would be ideal as it offers a single point of entry and makes it difficult for a user to bypass this check. This spot can be found within the arch/x86/entry/common.c as this is where all system calls will go through before being called. Depending on the implementation, a choice of x64 or x32 or even both system calls can be blocked by adding an if/else statement in the corresponding functions with a call to a getter function (explained next paragraph below). The if/else statement would just need to check the return from the getter function and see if the uid and syscall is found in the hash table. In the case of this project the function “do_syscall_64” was chosen as we are interested in blocking only x64 commands.

A file with helper functions would also need to be created or they can be added to the same file that the block/unblock system calls are put in. These helper functions would need to consist of a finder, getter and initializer. The finder would be called from the entry point of common.c, the finder function would receive a uid and syscall and then call the getter to get the hashtable needed. Then the function would check if the hashtable contains the uid in the table. The getter function would simply receive the system call and use the system call to return the array at the index (ie, array[systemcall]). The initializer function would be one time starter function for this feature. This function would be called from the block/unblock system call in the case that it is the first time that this feature is being called. Once called, the function will initialize a hashtable and have it point to the spot in the array and repeat until every spot of the array has a hashtable. To allow the block/unblock system calls be able to tell if the hashtables have been initialized, a global variable would be needed and initialized and only changed once initialize function is finished. By doing this, the operating system will save some memory as it won't have 436 hashtables until it is needed. Within each node of the hashtable, the node should have information of the uid and the amount of times the uid tried to access that system call when blocked.

*In the case the kernel hashtable implementation was not used, a simple hashtable can be created by using an array as the hashtable. Like before, helper 2 helper functions would need to be created which are, getHash, getTable. getHash would simply calculate the index of the array that the item will be placed in. The key can be easily calculated by taking the UID value of the user and taking the modulo of the array size to find the index. Meanwhile getTable will be a simple getter to get the hashtable of one of the 436 system calls, and just will need to return the hashtable array at any given array location back. One catch to this implementation would be a limitation of the initial array size, as to allow for constant access times, a fixed array would need to be used.

1.2. User-space Programs

For the userspace code, testing block can be done by calling block in the testing code and checking the return for a successful block operation. A more robust test however can be made by adding some lines to see if after a block, if the same block can be run again. If in the case the block is unable to run again, that means the block has been found in the hashtable and thus will be found later when blocking the syscall.

Likewise, for testing unblock, a similar method can be used in which a block command would need to be run before calling the unblock. This is the reason that the block command would need to be tested before testing the rest of the commands as the block allows successful operation of the other 2 sysalls. To test the unblock, a block command would need to be run to first block a call, then a unblock call can be used with the checking the return if the unlock command succeeded. To make the test more robust, after unblocking, a block command again can be run to see if the same call can be blocked again. If in the case the block goes though, then that means that the unblock call has successfully unblocked the command. Otherwise, the block will show that the block is unable to block again due to the unblock call not removing the block properly.

The last system call test would be for testing count. Since count is literally a counter for the amount of times a blocked used attempted to call a blocked system call, a block would need to be called and tested by making the test user call the blocked system all X amount of times and checking if the number adds up to the number of calls made.

In addition to the last three testers, while hard coding the syscall tests will be easy to do. It makes it harder to quickly test, thus it would be ideal to allow arguments as test values like ./test1 42 122. In this case the first argument will be the uid to block while the second one would be the syscall to block. By doing this, it will allow for quicker testing over multiple system calls for proper blocking. Some basic input checking should be added to check for invalid UID's like negative values.

A sbx421_run test should also be added which allows emulation of running an implemented syscall command as a given user with the second argument of a file of syscalls to block. This would allow testing for multiple users using the system calls for testing

2. References

"Basics of Hash Tables Tutorials & Notes: Data Structures." *HackerEarth*, https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/.

List any references you used in your assignment here. You do not have to list the Linux kernel source code files here (or this template), but you should list any web references or any other such things that you used in developing your assignment here. Please use a standard citation format (MLA or APA are fine) for your list of references.