CSC369 Assignment 1 - Hijacking System Calls and Monitoring Process

May 18, 2020

1 Overview

- In this assignment, you will achieve the goal of hijacking (intercepting) system calls by writing and installing a very basic kernel module to the Linux kernel.

  Here is what "hijacking (intercepting) a system call" means. You will implement a new system call named my_syscall, which will allow you to send commands from userspace, to intercept another pre-existing system call (like read, write, o pen, etc.). After a system call is intercepted, the intercepted system call would log a message first before continuing performing what it was supposed to do.

  For example, if we call my_syscall with command REQUEST_SYSCALL_INTERCEPT and target system call number __NR_mkdir (which is the macro representing the system call mkdir) as parameters, then the mkdir system call would be intercepted; then, when another process calls mkdir, mkdir would log some message (e.g., "muhahaha") first, then perform what it was supposed to do (i.e., make a directory).

  But wait, that's not the whole story yet. Actually we don't want mkdir to log a message whenever any process calls it. Instead, we only want mkdir to log a message when a certain set of processes (PIDs) are calling mkdir. In other words, we want to monitor a set of PIDs for the system call mkdir. Therefore, you will need to keep track, for each intercepted system call, of the list of monitored PIDs. Our new system call will support two additional commands to add/remove PIDs to/from the list.

  When we want to stop hijacking a system call (let's say mkdir but it can be any of the previously hijacked system calls), we can invoke the interceptor (my_syscall), with a REQUEST_SYSCALL_RELEASE command as an argument and the system call number that we want to release. This will stop intercepting the target system call mkdir, and the behaviour of mkdir should go back to normal like nothing happened.

2 Checklist

- Here is a checklist that should help get you started, and to make sure that you won't forget the important things:

  1. Get your starter code
  2. Test that you have access to the VM in the CDF labs (instructions below).
  3. Download the disk image for the virtual machine here (gzipped). On the host computer (your laptop or a lab computer), use a virtual machine software (VirtualBox or VMware) to create a virtual machine using the the disk image you downloaded (instructions to follow below).
  4. Read and understand the existing code in the starter code. This is an important step of this assignment, and you should not start writing your own code before you have a good understanding of the starter code.
  5. Implement the new kernel module by completing source file "interceptor.c". Sections that need to be completed are marked with the TODO tag). Do NOT modify the header file "interceptor.h".
  6. Make sure to test as you go. You should first make sure that the commands to intercept and de-intercept work well, before attempting to implement the monitoring commands.
  7. Testing and debugging (must be done in the virtual machine):
     (a) Check out your code inside the virtual machine.
     (b) Type make to compile your kernel module. Make sure there is no error or warning.
     (c) Implement the intercept and release commands.
     (d) Compile the test_intercept.c program using gcc.
     (e) Test your code using sudo ./test_intercept, and make sure that all tests pass.
     (f) Implement the monitoring/un-monitoring commands.
     (g) Compile the test_full.c program using gcc.
     (h) Test your code using sudo ./test_full, and make sure that all tests pass.
  8. Submit your code on time. See "Submission" for more details.
  9. Congratulations! You now have some great hands-on experience with the Linux kernel! You can now be proud of having programmed a Linux kernel module. You know what else are commonly implemented as kernel modules? Device drivers! Although they are more complex, you now technically have the basis to try to write one. Isn't that cool?

3 Goal

- The goal of this assignment is to learn more about system calls and to use synchronization mechanisms. For this assignment you will be writing a very basic kernel module that intercepts system calls and monitors processes on demand.

4 Requirements

- In order to be able to issue our own hijacking commands from userspace, we need a new system call that takes as parameters the command, the system call number (to be intercepted), and (for monitoring) a pid.

  Instead of adding a new system call, which can be tricky, our new system call my_syscall will be installed in place of an unused system call in the system call table. We will connect my_syscall to the entry number MY_CUSTOM_SYSCALL (in effect, entry 0 which is mostly unused). The new system call my_syscall, defined as follows: int my_syscall(int cmd, int syscall, int pid); will serve as an interceptor and will receive the following commands from userspace:

  - **REQUEST_SYSCALL_INTERCEPT:** intercept the system call syscall
  - **REQUEST_SYSCALL_RELEASE:** de-intercept the system call syscall
  - **REQUEST_START_MONITORING:** start monitoring process pid for system call syscall, i.e., add pid to the syscall's list of monitored PIDs. A special case is that if pid is 0 then all processes are monitored for syscall, but only root has the permission to issue this command (see the comments for my_syscall in the starter code for more details).
  - **REQUEST_STOP_MONITORING:** stop monitoring process pid for system call syscall, i.e., remove pid from the syscall's list of monitored PIDs.

  Kernel module operation

  Your kernel module must, upon initialization, replace the system call table entry for the MY_CUSTOM_SYSCALL number, with the my_syscall function. When the module is released, it must restore this system call to its original routine.

  As a result, when your kernel module is loaded, any subsequent invocations of the system call number MY_CUSTOM_SYSCALL from userspace, will issue four types of commands, to intercept or release a given system call, and to start and stop monitoring a pid for a specific syscall. You must implement the my_syscall function accordingly.

  1. **REQUEST_SYSCALL_INTERCEPT and REQUEST_SYSCALL_RELEASE**

     When an intercept command is issued, the corresponding entry in the system call table will be replaced with a generic interceptor function (discussed later) and the original system call will be saved. When a REQUEST_SYSCALL_RELEASE command is issued, the original saved system call is restored in the system call table in its corresponding position.

2. **2. REQUEST_START_MONITORING and REQUEST_STOP_MONITORING**

Monitoring a process consists of the module logging into userspace some information about the process and the system call: the system call number, the parameters of the system call, and the pid of the process.

When a REQUEST_START_MONITORING command comes through our custom system call, the kernel module must record internally that the pid passed as a parameter should be monitored for the syscall number (passed as a parameter as well). The monitoring can be done for a specific pid, or for all pids (in which case the pid parameter for my_syscall will be 0).

Ok, but I still don't understand, what does it mean to monitor a pid? And what does the generic interceptor function do?

Let's start with the monitoring. We have established that once the user issues a monitoring command, the kernel module records internally that pid should be monitored whenever it issues system call number syscall (it will be placed in a monitored list - see details in starter code).

We have also established that the generic interceptor function is what each intercepted system call will reach. In other words, whenever we reach the generic interceptor, we know that the system call is being intercepted (otherwise we would not reach this). If the pid of the process issuing the system call is being monitored, that means that we must print some information to a log. The log message will simply contain the system call number and the arguments, as well as the calling process's pid.

We have provided you in the starter code with a *log_message* macro, which takes care of sending a message to the system log. You can check the log using the *dmesg* command.

As you might expect, regardless if a pid is monitored or not, the generic interceptor must eventually (once it's done logging, if applicable), call the original system call to allow normal operation of all processes in the system.

Alrighte, but what if a process exits before the user can issue a system call to stop monitoring it?

Good question! When your kernel module initializes, you should also hijack the *exit_group* system call (with number *__NR_exit_group*), by replacing it in the system call table with your own custom function *my_exit_group*. Of course, make sure to save the original *exit_group* function, and to restore it when your kernel module is unloaded.

Implementing the *my_exit_group* function should be simple: all you have to do is to remove the pid of the exiting process from all kernel module's internal bookkeeping on monitored processes, then call the original *exit_group* function.

5 Error Conditions

- You must make sure to check any possible misuse of the commands. In case of a misuse, you should return a proper error code (e.g., -EINVAL, -EPERM, google "Linux error code" for more information on error codes). Here is a list of things you should keep in mind:

A. For each of the commands, check that the arguments are valid (-EINVAL): The syscall number must be valid: not negative, not ¿ NR$_s yscalls(thelastsyscallnumberinthetable), a$
$pid_t ask(find_v pid(pid), PIDTYPE_P ID) Checkthatthecalledhastherightpermissions(-EPER$
$Forthefirsttwocommands, wemustberoot(seethecurrent_u id()macro), tobeabletointerceptorrele$
$Isthecallingprocessroot?ifso, allisgood, nodoubtsaboutpermissions.Ifitisnot, thencheckifthep$

B. Check for correct context of commands (-EINVAL): Cannot de-intercept a system call that has not been intercepted yet. Cannot stop monitoring for a pid that is not being monitored, or if the system call has not been intercepted yet. If the system call has not been intercepted yet, a command to start monitoring a pid for that syscall is also invalid.

C. Check for -EBUSY conditions:
   - If intercepting a system call that is already intercepted.
   - If monitoring a pid that is already being monitored.

D. If a pid cannot be added to a monitored list, due to no memory being available, an -ENOMEM error code should be returned. The starter code provides a set of functions that enable operation with kernel lists. What if a stop monitoring request comes in for a specific PID (let's call it P), for a syscall that monitors all PIDs? Is that an error or should we treat this as a special case? The answer is the latter, it should be treated as a special case, do not return an error code. If we already monitor all PIDs for a syscall, then you might have to think of a solution to make sure that you can keep monitoring all the PIDs in the system, except for P. Please keep in mind that some processes that will be monitored may not have even started their execution. Also, please keep in mind that we might have other stop monitoring requests for the same syscall, in which case, you might have to think of how to use the list of monitored pids in a smart way. One possibility is turning the list of monitored pids into a "blacklist" (keeping track of the pids that are not being monitored).

6 General Information