Assignment 1 - Hijacking System Calls and Monitoring Processes

Due: Jan 27, at 10:00pm

Overview

In this assignment, you will ac[illegible]ijacking (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, open, 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.

Checklist

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

1. Find your SVN repository on MarkUs (see "Submission"), and do a checkout to make sure you can access it. The starter code files can be found on the teaching lab machines (either the servers or workstations), under /u/csc369h/winter/pub/a1-starter/starter_code.tgz
2. Test that you have access to the VM in the teaching labs (instructions below).
3. Download the disk image for the virtual machine here (gzipped) (http://www.cs.toronto.edu/~bogdan/stuff/UbuntuServer-CSC369-upd.zip).
   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 [illegible] **the virtual machine**):
    a. Check out your [illegible]ual machine.
    b. Type `make` to co[illegible]module. Make sure there is no error or warning.
    c. Implement the in[illegible]e commands.
    d. Compile the test[illegible]m 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?

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.

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:

a. REQUEST_SYSCALL_INTERCEPT: intercept the system call `syscall`
b. REQUEST_SYSCALL_RELEASE: de-intercept the system call `syscall`
c. 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).
d. 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 star[illegible]ing a pid for a specific syscall. You must implement the `my_syscall` function accord[illegible]

1. REQUEST_SYSCALL_INT[illegible]QUEST_SYSCALL_RELEASE.
When an intercept command [illegible]esponding entry in the system call table will be replaced with a generic interceptor fun[illegible]ter) 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. 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.

Alright, 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.

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_syscalls (the last syscall number in the table), and n[illegible]SYSCALL itself (for obvious reasons).
   - The pid must be [illegible]oring commands. It cannot be a negative integer, and it must be an exist[illegible] the case when it's 0, indicating that we want to start/stop monitoring for **al**[illegible]
     If a pid belongs t[illegible] then the following call is not NULL:
     `pid_task(find_vpid(pid), PIDTYPE_PID)`

B. Check that the called has the right permissions (-EPERM):
   - For the first two commands, we must be root (see the current_uid() macro), to be able to intercept or release system calls.
   - For the last two commands, the following logic applies:
     - Is the calling process root? If so, all is good, no doubts about permissions.
     - If it is not, then check if the `pid` requested is owned by the calling process
     - Also, if `pid` is 0 and the calling process is not root, then access is denied (monitoring all pids should only be allowed for a superuser, for obvious reasons).

C. 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.

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

E. 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? For this assignment, you can assume that this is an error and simply return -EINVAL.
BONUS (5% applicable to marks lost elsewhere on this assignment): Instead of ignoring this case and returning -EINVAL, you can optionally treat this as a special case. 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).

General information

1. You must use the starter code provided, which gives you detailed instructions on what you need to implement. Please make sure to implement all the parts indicated using detailed TODO comments. Please make sure to first attend the tutorial which will help you write a simple kernel module and show you how to use printk statements for debugging. See the tutorial notes as well.
2. Your assignment will be tested on a virtual machine on CDF (aka teaching labs). You can access this virtual machine from yo ccount, or you can download the provided virtual machine disk image and install i computer through a virtualization solution (for example, free software include V ualBox, etc.)
3. We strongly recommen use the virtual machine for development, but rather only for testing and debugg on this assignment, it is quite likely you will crash the kernel and although yo rt the VM, there will be no guarantee that your code will still be there (the VM tools on the teaching labs won't guarantee you safe snapshots). To prevent your hard work from possible data corruption, either do an SVN checkout inside the VM and use your repository to commit your code periodically from within the VM, or make sure to at least back up your code periodically (e.g., by scp-ing it back to your teaching labs account).

Accessing the Virtual Machine on the teaching labs

Guidelines for accessing the VM on the teaching labs can be found here (VirtualMachineInstructions.txt). Please make sure to follow the instructions carefully.

Setup VM On Your Own Machine

Note: Your assignment has to ultimately be tested on a teaching lab machine. However, if you wish to develop it and test it first on your own machine, using virtualization software (*do not test your assignment directly on your computer!*), then we will provide some basic instructions on how to do so. Since VirtualBox is one of the most portable (as well as free) virtualization software, here (VirtualBoxInstructions.html) and here (http://www.cs.toronto.edu/~dbkats/csc369-virtualbox-instructions.html) are some basic guidelines on how to install the VM image in VirtualBox on your computer (of course, many tutorials can be found online as well, so feel free to consult other sources if something does not work well for your own machine).

Implementation details

1. Since the number of system calls is rather small (~300), and for performance reasons, you must maintain the system call information in an array. Each array element will contain information, as described in the mytable struct:
```
typedef struct {
    asmlinkage long (*f)(struct pt_regs);
    int intercepted;
    int monitored;
    int listcount;
    struct list_head my_list;
} mytable;
```
2. You must use a linked list for storing information about the monitored processes; using an array of

fixed size is entirely inadequate (because the search time will be the same as a linked list, the implementation complexity will be the same, but the size of the array will limit the number of entries).

3. The system call table is exported by the void* sys_call_table[NR_syscalls], present in one of the kernel source files from the VM image on the teaching labs. If you wish to configure your own kernel image and re-compile it, you can modify the source code by adding the following two lines in the `/usr/src/linux-source-2.6.32/arch/x86/kernel/i386_ksyms_32.c` file:
   ```
   extern void* sys_call_table[];
   EXPORT_SYMBOL(sys_call_table);
   ```
   then recompile the kern[illegible]al machine image already has these changes in place.
4. Since the 2.6 kernel is [illegible]ust protect access to shared data. You will be using spinlocks for this purp[illegible]nlocks is fairly simple and you have been shown some examples in one of the [illegible]
5. You must use the syste[illegible]r MY_CUSTOM_SYSCALL. Do not attempt to use a different existing system call number, as that may result in the kernel misbehaving (to say the least). Remember that lots of services running in your OS make use of these system calls.
6. Logging the system call will be done using the `log_message` macro, defined in the interceptor.h header file.
7. For testing, you can use the provided tester programs. After you compile a test program (the provided Makefile only compiles your interceptor module, not any tester), remember to run the tester using sudo privileges in the VM.
   To facilitate your testing, you should first try to implement the commands to intercept and release system calls. When you are ready to test these, use the `test_intercept.c` tester.
   Once all tests pass, you can proceed to implementing the monitoring commands. To test all commands (both related to intercepting and to monitoring), you can use the `test_full.c` tester.

Testing your code

To help you test your code, we have provided two testers, which you will also find in your repositories. To encourage you to test as you go, we are providing you with two testers:

- `test_intercept.c` - tests if your intercept and de-intercept commands work correctly. You should first implement these and make sure the tester passes all cases.
- `test_full.c` - tests if all commands (including intercept, release, and both monitoring commands) work properly. This is a superset of the first tester, and you should only use once your code passes the first tester.

The tester loads your module and tests some basic functionality. It is by no means a comprehensive tool to ensure your code works for every corner case. To ensure that your code works correctly in all possible scenarios, you should add more test cases by modifying the testers (see code comments in main). However, please do not submit your own tester files, because they will not be marked. The tester will also not catch synchronization bugs, except for blatant deadlocks. It is your responsibility to ensure that your code is not likely to run into synchonization problems. Finally, when testing, you will likely see the tester crash on various tests, due to bugs in your module. During your debugging, please feel free to go in each tester, and comment out some of the system calls being tested, if you wish to debug each test case in isolation.

Other Useful Tips

- Again, run tests ONLY in the virtual machine, NOT native computer, unless you hate your laptop.

- Once more, we strongly recommend that you do NOT use the virtual machine for development, but rather only for testing and debugging. Since it is quite likely you will crash the kernel and there will be no guarantee that your code will be intact. To prevent your hard work from possible data corruption, either do an SVN checkout inside the VM and commit your code periodically, or make sure to at least back up your code periodically.
- Reading and understanding code is as important as (if not more important than) writing code.
- The comments in the starter code have a lot of information, make sure to read them carefully.
- Remember that when we de-intercept a syscall, the original system call must be restored in the system call table. For this, make sure to properly store the original system call before replacing it.
- For debugging, learn how to use the `printk` function, which prints messages to kernel log. See tutorial notes as well.
- Use `dmesg` command to view the kernel log.

Submission

You will submit the `interceptor.c` file that contains your implementation, along with the files required to build your program (including the provided `interceptor.h`, `Makefile`, and `Kbuild`, which you should not modify). Do not submit executables, or tester files!

For those working in pairs, please make sure to commit to the group repository. If you previously had trouble forming groups on A0, please contact me in advance. Do not leave this to the last minute, technical trouble with your repository will not get you an extension!

Additionally, you must submit an INFO.txt file, which contains as the first 3 lines the following:

- your name
- your UtorID(s)
- the **svn revision number** for your last submission. As a general rule, we will always take the last revision before the deadline (or after, if you decide to use grace tokens), so this is simply a sanity check for us that we did not miss a revision when we retrieve your code via MarkUs.

Aside from this, please feel free to describe problems you've encountered, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken, etc.

Make sure your code compiles without any errors or warnings.
Code that does not compile will receive zero marks!

Marking scheme

We will be marking based on correctness (90%), and coding style (10%). Make sure to write legible code, properly indented, and to include comments where appropriate (excessive comments are just as bad as not providing enough comments). Code structure and clarity will be marked strictly!
Once again: code that does not compile will receive 0 marks!
