CIS–4020 Lab
Counting System Calls

© Copyright 2016 by Peter C. Chapin

Last Revised: August 11, 2016

1 Introduction

In this lab you will get some experience modifying the Linux kernel, writing a kernel module, and exploring the kernel source code in general using tools like `cscope`.

Consider the following problem: Suppose you wanted to count the number of times each system call on your system is invoked. A program like `strace` can show you all the system calls made by a particular process, but it would be infeasible to use a tool like `strace` to examine all processes that ever execute on your system. However the kernel is aware of all system calls (of course). Thus one approach to counting system calls would be to modify the kernel to gather that information. This is the approach you will take in this lab.

Counting system calls is of little value if there is no way to read the resulting data. Thus you will also need to provide a mechanism that allows an ordinary process to extract the current counts from the kernel. There are several possible ways to do this. For example you could add a new system call or you could provide a driver with a suitable `ioctl` interface. For this lab you will use a simpiler and more appropriate method. You will write a module that implements a file in the `/proc` file system such that reading the file returns the various counts.

2 Counting Calls

In the first part of this lab you need to modify the kernel so that it counts the number of times each system call is invoked. This is not as hard to do as it may sound. System calls all pass through an entry point in the file `arch/x86/kernel/entry_64.S`, starting at the label `system_call` in that file. The value in `rax` at that time is the system call number of the call being made. The kernel also supports other entry points. For example, system calls made from 32 bit code or invocations of fault handling routines have separate entries. It would be an interesting extension to this lab to count the usage of those other entries as well.

To count the system calls you must declare an array of 64 bit unsigned long integers large enough to hold a counter for every possible call. Just before the system call is dispatched, use the value in `rax` to increment the appropriate counter in the appropriate array. Note that if you make your array global it will automatically be zeroed early in the kernel's boot sequence. Thus you don't need to worry about explicitly initializing the array.

You will need to export the array so that they will be visible to your module. You may also want to define and export a single variable containing the number of system calls so that your module knows how large the array is.

After making these changes you should recompile the kernel and then reinstall it as the boot kernel on HackBox. Reboot the machine to verify that the kernel still works. I recommend that you create a snapshot before making these changes.

3 Displaying the Counts

Now that the kernel is counting system calls, you need to write a module that accesses that information and provides a way for normal processes to read it.

Start with the by building the `/proc` file skeleton provided on the class's web site. Read the comments in that file to learn how to use the `seq_file` API. You may also want to read the source of the `seq_read` function in the kernel. That function coordinates the activity of the functions you must provide; reading `seq_read` allows you to see how your functions will be used. If you search online for information about the `seq_file` API be aware that some descriptions are inaccurate and incomplete; proceed with open eyes.

For each system call output the counts with one on each line. I suggest a format such as follows.

```
nnn: xxxxxxxxxxxxxxxxxxxx
```

Here `nnn` is a three digit system call number and the counts are displayed as 20 digit numbers to allow for their maximum (64 bit) value. The `seq_printf` function behaves much like the standard `printf` function. Consult information about `printf` to see how to format integers into fixed width fields. Aligning the output nicely makes the table easier to read.

Ideally the output would include the system call names instead of the numbers since very few people are familiar with the numbers. However, for this lab it is sufficient to print the call numbers. You can leave the conversion of number to name to a higher level tool (perhaps a GUI tool) written by someone else. This is actually quite appropriate. Many `/proc` files produce very raw output.

4 Report

Write a report for this lab following the lab report template provided. Desribe how your code works. Comment on the behavior of the counts and discuss any issues you see. Do you see anything unusual in the way the counts behave as the system runs? Include "significant" code fragments in your report but you don't need to include complete listings.