CS 3013 Operating Systems
Craig E. Wills
Assigned: Tuesday, September 3, 2019

WPI, A Term 2019
Project 2 (20 pts)
Due: Friday, September 13, 2019

Project 2: Kernel Interception

Loadable Kernel Modules (LKMs) allow you to change the operation of the base operating system arbitrarily. As noted in Pre-Project 2, LKMs are not able to add system calls, but they **are** able to change existing system calls. This process is known as a system call "interception." In this project, we will perform two types of system call interceptions: a straightforward monitoring approach and a more creative re-interpretation of an existing system call.

Part 1: Keeping Up with the Users

Goal: As a warm-up, we'll make some simple alterations to existing system calls. Every time a regular user opens or closes a file, we will print this information to the system log. However, we do not want this to happen for non-user actions (such as those by the root user, known service user accounts, etc.). Accordingly, we need to intercept and modify the existing system calls for `open` and `close` to add `kprint` statements for regular users. In the syslog, the `open` messages should look like

```
Sept 6 18:24:52 dalek kernel: [ 105.033521] User 1000 is opening file: /etc/motd
```

while the close messages may look like

```
Sept 6 18:24:53 dalek kernel: [108.511234] User 1000 is closing file descriptor: 2
```

While it is up to you to intercept `open` and `close`, we will provide an example of intercepting one of the system calls we made in Pre-Project 2 to give you an idea of the mechanisms available to you. The following is a full example module that does an interception. After this code listing, we provide commentary on how it works (this code is available on the course Web site as `mymodule.c`).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>

unsigned long **sys_call_table;

asmlinkage long (*ref_sys_cs3013_syscall1)(void);

asmlinkage long new_sys_cs3013_syscall1(void) {
    printk(KERN_INFO "\"'Hello world?!' More like 'Goodbye, world!' EXTERMINATE!\" -- Dalek
    return 0;
}

static unsigned long **find_sys_call_table(void) {
  unsigned long int offset = PAGE_OFFSET;
  unsigned long **sct;

  while (offset < ULLONG_MAX) {
    sct = (unsigned long **)offset;

    if (sct[__NR_close] == (unsigned long *) sys_close) {
      printk(KERN_INFO "Interceptor: Found syscall table at address: 0x%02lX",
                (unsigned long) sct);
      return sct;
    }
```

```
    offset += sizeof(void *);
  }

  return NULL;
}

static void disable_page_protection(void) {
  /*
    Control Register 0 (cr0) governs how the CPU operates.

    Bit #16, if set, prevents the CPU from writing to memory marked as
    read only. Well, our system call table meets that description.
    But, we can simply turn off this bit in cr0 to allow us to make
    changes. We read in the current value of the register (32 or 64
    bits wide), and AND that with a value where all bits are 0 except
    the 16th bit (using a negation operation), causing the write_cr0
    value to have the 16th bit cleared (with all other bits staying
    the same. We will thus be able to write to the protected memory.

    It's good to be the kernel!
   */
  write_cr0 (read_cr0 () & (~ 0x10000));
}

static void enable_page_protection(void) {
  /*
   See the above description for cr0. Here, we use an OR to set the
   16th bit to re-enable write protection on the CPU.
  */
  write_cr0 (read_cr0 () | 0x10000);
}

static int __init interceptor_start(void) {
  /* Find the system call table */
  if(!(sys_call_table = find_sys_call_table())) {
    /* Well, that didn't work.
       Cancel the module loading step. */
    return -1;
  }

  /* Store a copy of all the existing functions */
  ref_sys_cs3013_syscall1 = (void *)sys_call_table[__NR_cs3013_syscall1];

  /* Replace the existing system calls */
  disable_page_protection();

  sys_call_table[__NR_cs3013_syscall1] = (unsigned long *)new_sys_cs3013_syscall1;

  enable_page_protection();

  /* And indicate the load was successful */
  printk(KERN_INFO "Loaded interceptor!");
```

```
  return 0;
}

static void __exit interceptor_end(void) {
  /* If we don't know what the syscall table is, don't bother. */
  if(!sys_call_table)
    return;

  /* Revert all system calls to what they were before we began. */
  disable_page_protection();
  sys_call_table[__NR_cs3013_syscall1] = (unsigned long *)ref_sys_cs3013_syscall1;
  enable_page_protection();

  printk(KERN_INFO "Unloaded interceptor!");
}

MODULE_LICENSE("GPL");
module_init(interceptor_start);
module_exit(interceptor_end);
```

Understanding the Example

While a useful example, many parts may not be clear yet. Let's talk about the code in pieces:

```
unsigned long **sys_call_table;

asmlinkage long (*ref_sys_cs3013_syscall1)(void);
```

The first line hints that we are going to need to find the system call table in memory (using some cute pointer tricks). We are going to use `find_sys_call_table` to find this value, and then change some of the existing system call pointers to new functions in the `interceptor_start` function.

The second line is going to be a variable that holds the pointer to the existing `cs3013_syscall1` function. We are going to intercept this call and replace it, but when we unload the module (in `interceptor_end`), we will want to restore the original. This variable will let us keep tabs on the original so we can safely restore the old system state. **You will want to be certain to do this right when intercepting `open` and `close`.**

Now, let's consider our first function:

```
asmlinkage long new_sys_cs3013_syscall1(void) {
    printk(KERN_INFO "\"'Hello world?!' More like 'Goodbye, world!' EXTERMINATE!\" -- Dale
    return 0;
}
```

This represents what we are replacing our original system call. While our `cs3013_syscall1` previously cheerfully shouted "Hello, world!", our injected system call will be slightly more menacing. At least it will be easy for us to tell them apart. The beginning portion, `asmlinkage long` will be consistent for all the system calls you'll want to intercept.

The `find_sys_call_table` function is not something you will need to modify, but it will be exceedingly useful. In the latest versions of Linux, the kernel developers stopped exporting the symbol to tell you where the system call table is. However, some system calls have to be exported, in particular, `sys_close`. So, the `find_sys_call_table` function scans memory looking for the pointer that matches the pointer to the `sys_close` system call. Since it knows that `__NR_close` is index into the system call table (since system calls are numbered consecutively), it can find the memory address of the beginning of the system call table. Nifty, eh?

The `disable_page_protection` and `enable_page_protection` are functions that invoke a routine that will disable the protection for read-only memory on the processor, allowing us to overwrite the system call table, even though that would normally be forbidden. You are welcome to reuse these functions. To use them, you basically want to disable the protections before a system call table modification and then immediately re-enable the protections. If

you forget to re-enable the protection, misbehaved processes will suddenly be able to modify pages that should be protected. This can cause memory issues that are very hard to debug. You've been warned.

The `interceptor_start` function finds the system call table, saves the address of the existing `cs3013_syscall1` in a pointer, disables the paging protections, replaces the `cs3013_syscall1`'s entry in the page table with a pointer to our new function, then re-enables the page protections and prints a note to the kernel system log.

The `interceptor_end` function essentially reverts the changes of the `interceptor_start` function. It uses the saved pointer value for the old `cs3013_syscall1` and puts that back in the system call table in the right array location.

When you go to add your interceptors for the `sys_open` and `sys_close` calls, you will need to place them in the same place as the interceptors for `cs3013_syscall1` function. You'll then write your own version of these system calls (equivalent to our `new_sys_cs3013_syscall1` function). Make sure you replicate the parameters of the `sys_open` and `sys_close` calls.

Helpful Hints:

- Copy and modify your `Makefile` from the pre-project to build the `module.ko` module from the `mymodule.c` file.
- Test out the new `cs3013_syscall1` before writing your own code to make sure you understand how to do injections properly. Remember that you can insert modules with `sudo insmod module.ko` and remove them with `sudo rmmod module.ko`. Keep the `cs3013_syscall1` interception around as an example for Part 2.
- When intercepting `sys_open` and `sys_close`, you can invoke the old versions of these calls by using the reference pointers you saved to restore the system calls. As a reference, the instructor's code for `new_sys_open` function was 4 lines of code. Do not make this step harder than it needs to be!
- The `current_uid()` function will return the account number of the currently running user. In Ubuntu, regular user accounts start at UID 1000.
- This may sound silly, but to test that the monitoring code is tracking users properly, you will need to open/close files as a regular user and not as `root` or via `sudo`.

Part 2: Getting Process Information

You will now create a new system call that gets some useful information about the current process, and you will add it to your kernel by using a Loadable Kernel Module (LKM) to intercept a system call created in Pre-Project 2. In particular, you will intercept `cs3013_syscall2`. Note that in Pre-Project 2, this system call did not take any parameters. But, with our interception, we can redefine the function to take a pointer as a parameter. Naturally, this will break your `testcall` code from Pre-Project 2. This shows you both the power of LKMs and shows you why kernel developers usually do not redefine system calls. After all, how many `wait` system calls are there?

The function prototype for your system call will be

```
long cs3013_syscall2(struct processinfo *info);
```

where `*info` is a pointer to data structure in user space where your system call will put information about the process. The system call returns zero if successful or an error indication if not successful.

The structure `processinfo` is defined as follows:

```
struct processinfo {
    long state;               // current state of process
    pid_t pid;                // process ID of this process
    pid_t parent_pid;         // process ID of parent
    pid_t youngest_child;     // process ID of youngest child
    pid_t younger_sibling;    // pid of next younger sibling
    pid_t older_sibling;      // pid of next older sibling
    uid_t uid;                // user ID of process owner
```

```
    long long start_time;     // process start time in
                              // nanoseconds since boot time
    long long user_time;      // CPU time in user mode (microseconds)
    long long sys_time;       // CPU time in system mode (microseconds)
    long long cutime;         // user time of children (microseconds)
    long long cstime;         // system time of children (microseconds)
};    // struct processinfo
```

If the calling process does not have, say, a youngest child or any siblings, then return `-1` in the corresponding fields. In Linux, children and siblings are not stored in any particular order. Therefore, the youngest child is the process in the `children` list with the latest `start_time`. Likewise, the next younger sibling is the process in the siblings list with a start-time that is larger than, but nearest to, the `start_time` of the current process. Likewise, the next older sibling is the one in the list with a next smaller `start_time`.

You will need to have a copy of the `processinfo` structure definition in both the interceptor C code and the user-space application that invokes it. Feel free to include it using header files if desired. However, be aware that you may need to use different `#include` functions to access the data types referenced in the structure in the kernel module verses the user space application.

To help you test, note that some of the information returned by `getprocessinfo()` is the same as the information you obtained in Project 1 using `getrusage()`. You might compare the two results to make sure your code is right.

Implementation

Before you attempt to implement your system call, you should look at the implementations of simple system calls, such as `getuid` and `getpid`, to provide guidance. These can be found in the file `kernel/sys.c`. Here are some things you should know:

- Almost all of the information you need to fill in the fields of a `processinfo` structure can be found in the structure called `task_struct`, defined in `include/linux/sched.h` in the kernel source tree. Study this structure carefully!

  Some of the information is obtained by following pointers or doubly-linked lists from `task_struct` – for example, child or sibling processes. If a process has no children or siblings, these lists will be empty. You need to use the linked list macros described in Chapter 6 of Linux Kernel Development, 3rd edition, and `linux/list.h` to access them.

- The kernel file `include/asm/current.h` defines an inline macro called `current` that returns the address of the `task_struct` of the current process.

- Most of the fields of `task_struct` specify time values in an internal format called *jiffies*, as discussed in Chapter 11 of Linux Kernel Development, 3rd edition. These are fields of type `cputime_t`, and a kernel macro called `cputime_to_usecs()` converts these time values into microseconds. This can be found in `include/asm-generic/cputime.h`. There are also macros for adding, subtracting, and comparing `cputime_t` values.

  The field of the `task_struct` that specifies the start time of the process is a struct called `timespec`, that specifies time in seconds and nanoseconds. A macro in the file `linux/time.h` called `timespec_to_ns()` converts a `timespec` into a number of nanoseconds.

- Every system call must check the validity of the arguments passed by the caller. In particular, kernel code must never, ever blindly follow pointers provided by a user space program. Fortunately, the Linux kernel provides two functions that check the validity and also transfer information between kernel and user space. These functions are `copy_from_user` and `copy_to_user`, and they are defined in `include/asm-generic/uaccess.h`. You will need to use the latter. See pp. 76-77 in Linux Kernel Development, 3rd edition.

  For example, suppose you have accumulated information in a kernel data structure called `kinfo`, then you can use `copy_to_user` as follows:

```
/* copy data from kinfo to area in user space pointed to by
   ''info'', a pointer supplied by caller */
if (copy_to_user(info, &kinfo, sizeof kinfo)
   return EFAULT;
```

where `EFAULT` is an error code.

The `copy_to_user` function returns zero if the info argument provided by the caller is valid and the copy is successful, but it returns the number of bytes that failed to copy in case of an error.

- You dont need to worry about page faults in the user space or about blocking and/or pre-emption by another process. Your system call operates in process context, which is essentially an extension of the user-space process. It has access to both kernel and user data, and it is capable of taking page faults, being pre-empted, or going to sleep without affecting the kernel or other processes.
- You need to intercept the `cs3013_syscall2` system call, as you did for `cs3013_syscall1` in Part 1 of this project.

Hint: Start out your `cs3013_syscall2` system call with getting just a few pieces of information, so as to make sure that you can return it to the caller. After you get this part working, add the functionality to access the `task_structs` of the parent, child, and siblings. Remember that the children and siblings are in linked lists that need to be accessed using the Linux kernel list macros described in Chapter 6 of the Love text.

Testing your System Call

Write a user-space test program that calls `cs3013_syscall2` patterned after the one you wrote for testing your `cs3013_syscall1` call in Part 1. This test program should include the user-space version of the `processinfo` structure described above. Your test program should print all of the information returned in the `processinfo` structure for the calling process. Run it several times, and run it from several different shells. Also test what happens if the pointer argument is null or invalid; show that the system call does the right thing.

Note which fields change from run to run and which fields do not. Also, inspect the results to see if they make sense. In your write up, discuss why some things change and how frequently they change.

For debugging your system call, use the `printk()` function that you used in Part 1. You may see this information in the `/var/log/syslog` file or using the `dmesg` command. Note that `dmesg` outputs these `printk` messages immediately while the `syslog` daemon uses buffering.

Deliverables and Grading

When submitting your project (only one team member needs to submit), please include the following (with all team member names on each file):

- The source code for the kernel module used for Phase 1 and Phase 2. Include header files, if needed.
- The user-land source code to test Phase 1 and Phase 2. For Phase 1, include the test calls to `cs3013_syscall1`. For Phase 2, include the code to test `cs3013_syscall2`.
- The Makefiles for the LKM and for the user-land testing code.
- Appropriate portions of the `/var/log/syslog` file for Phase 1 and Phase 2. For Phase 1, the log should include output of all the notifications you have of user activity.

Please compress all the files together as a single .zip archive for submission. As with all projects, please only standard zip files for compression; **.rar, .7z, and other custom file formats will not be accepted**.

Please upload your .zip archive via InstructAssist with the project name of *proj2*.