CS 3013 - Operating Systems

Project 2 (100 points)
Assigned: Friday, January 25, 2019
Checkpoint: Friday, February 1, 2019
Due: Tuesday, February 5, 2019

Project 2: Kernel Interception

Loadable Kernel Modules (LKMs) allow you to change the operation of the base operating system arbitrarily. As noted in Project 0, 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 reinterpretation of an existing system call.

You may use the same virtual machine you created in project 0; However, if you are have trouble getting the example code to work, then you should instead use the virtual machine image provided here: `http://cerebro.cs.wpi.edu/cs3013/CS%203013%20Ubuntu.vdi`. When booting, make sure to select the kernel ending with the string `cshue` and use the password `password`.

Part 1: On-Access Anti-Virus Scanner

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 `Jan 6 18:24:52 dalek kernel: [ 105.033521] User 1000 is opening file: /etc/motd`, while the close messages may look like `Jan 6 18:24:53 dalek kernel: [ 108.511234] User 1000 is closing file descriptor: 2`. We will also look at every `read` call to determine if the file contains the string `VIRUS`. If it does, we'll write a warning to the system log: `Jan 6 18:24:52 dalek kernel: [ 105.033521] User 1000 read from file descriptor 2, but that read contained malicious code!`. Hopefully, the user will be appropriately terrified by this message.

While it is up to you to intercept `open`, `close` and `read`, we will provide an example of intercepting one of the system calls we made in Project 0 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.

```
#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`, `close`, and `read`;** otherwise, a reboot will likely be in your future.

Now, let's consider our first function:

```
asmlinkage long new_sys_cs3013_syscall1(void) {
    printk(KERN_INFO "\"'Hello world?!' More like 'Goodbye, world!' EXTERMINATE!\" -- Dalek");
    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 reenables 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`, `sys_close`, and `sys_read` 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`, `sys_close`, and `sys_read` calls.

Helpful Hints:

- 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. Since it can be a pain to copy/paste from PDF documents, we have shared the original code at `http://cerebro.cs.wpi.edu/cs3013_project2.c`
- When intercepting `sys_open`, `sys_close`, and `sys_read`, 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 obvious, 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` (since that changes the UID).

Part 2: Process Genealogy

We are going to extend the Linux kernel to allow us to learn more about processes. We're going to learn about a targeted process's ancestors, siblings, and children.

In this phase of the project, we will create a new executable named `procAncestry`. The `procAncestry` command will take one parameter: the process ID (PID) to examine. The `procAncestry` command will go through all the processes running on the system and find the targeted PID. Once it does, it will traverse

the process's children and siblings, noting the PIDs for each. Once that's done, it will begin climbing the process tree ancestry, beginning with the target process's parent, followed by it's grandparent, and so on until it reaches a parentless node (likely init).

Implementation

You will need to create code, both in the kernel and in user space, to implement this executable. You will add the kernel code using a Loadable Kernel Module (LKM) to intercept two of the system calls created in Project 0. In particular, you will intercept `cs3013_syscall2` to implement the `procAncestry` functionality. Note that in Project 0, this system call did not take any parameters. But, with our interception, we can redefine the function to take pointers as parameters. Naturally, this will break your `testcall` code from Project 0. 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 call variants are there?

The function prototype for your system call will be `long cs3013_syscall2(unsigned short *target_pid, struct ancestry *response);`, where `*target_pid` is a pointer to an unsigned short and response is a pointer to an ancestor struct, as defined below. Both of these variables must have their memory allocated in user space before invoking the system call, otherwise an error will be returned. The system call returns zero if successful or an error indication if not successful. The system call will read the `target_pid` variable and will search for the process associated with that process ID. The kernel must print a message about the target process, the siblings, children, and ancestors in the system log, indicating what relationship the process has to the targeted process.

```
struct ancestry {
   pid_t ancestors[10];
   pid_t siblings[100];
   pid_t children[100];
};
```

Here are some things you should know:

- Almost all of the information you need 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, parent, 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. This gives you a place to start your iteration.

- 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 both. 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 FAULT;
  ```

where `EFAULT` is a error code defined in `include/asm/errno.h`.

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 do not 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 Calls

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.

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.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any assignment in which Part 1 is completed will receive full credit towards the checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting your project, please include the following:

- 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 source code for the executables to invoke `cs3013_syscall2` and the test program that you used to confirm proper user switching.
- The Makefiles for the LKM and for the user-land testing code.
- 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. For Phase 2, you should see the required messages about the process IDs as they are being checked.

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**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://ia.wpi.edu/cs3013/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://ia.wpi.edu/cs3013/files.php`),
3. Complete your Partner Evaluation (URL: `https://ia.wpi.edu/cs3013/evals.php`), and
4. Schedule your Project Demonstration (URL: `https://ia.wpi.edu/cs3013/demos.php`), which may be posted slightly after the submission deadline.

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback.

Groups **must** schedule an appointment to demonstrate their project to the teaching assistants. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, he or she will receive a 10 point reduction on his or her project grade.

During the demonstrations, the TAs will be evaluating the contributions of group members. We will use this evaluation, along with partner evaluations, to determine contributions. If contributions are not equal, under-contributing students may be penalized.

Project 2 – Kernel Modifications – Grading Sheet/Rubric

				Evaluation?
Grader:		Student Name:		
Date/Time:		Student Name:		
Team ID:		Student Name:		
Late?:				
Checkpoint?:			Project Score:	

Earned	Weight	Task ID	Description
_____	5%	1	Part 1 – Correct interception of cs3013_syscall1.
_____	10%	2	Part 1 – Correct interception/modification of open/close system calls.
_____	5%	3	Part 1 – Open/close log output for regular users only. Prerequisite: Task 2.
_____	5%	4	Part 1 – Correct user-side program for testing. Prerequisite: Task 1.
_____	25%	5	Part 2 – Correct kernel-side implementation of cs3013_syscall2 for user-kernel communication. Prerequisite: Tasks 1-4.
_____	20%	6	Part 2 – Correct kernel-side implementation of list-functions (children, sibling traversals) for cs3013_syscall2. Prerequisite: Task 5.
_____	15%	7	Part 2 – Correct process ID tracking/printing to syslog. Prerequisite: Task 5.
_____	15%	8	Part 2 – Correct user-side applications, test program, and exhaustive test results showing correct results. Prerequisite: Task 7.

Grader Notes: