

Lab 6

Loadable Kernel Modules (LKM) and Device Drivers

ITSC304: Operating Systems Exploitation

Table of Contents

[Lab Outcome(s) 2](#_Toc477907328)

[Reading 2](#_Toc477907329)

[Introduction 2](#_Toc477907330)

[1.0 Analyze and create Linux Loaded Kernel Module (LKM) 3](#_Toc477907331)

2.0 Create Linux Character Device Driver Module .....……………..………………………….….8

3.0 Windows Device Driver……………….……..……………………….…………………………13

L*abs must be submitted by the due date for full credit. After due date late submissions will be accepted for a period of one week (seven days) and the grade will be reduced by ten percent (10%) per day after due day.* ***Assignments that are submitted more than seven days late will receive a grade of zero (0).***

I certify that the work submitted in this assignment is my own and that it has not been taken in whole or in part from any other source. I understand that the penalty for plagiarism will include a grade of zero (0) for this assignment plus disciplinary action in accordance with SAIT policies.

**EVALUATION**

|  |  |  |
| --- | --- | --- |
| Analyze and Create Linux LKMs | 20 |  |
| Modify Character Device Driver Module | 23 |  |
| Windows Device Driver | 12 |  |
| TOTAL MARK | 55 |  |

Lab Outcome(s)

* .Analyze and create Linux Loadable Kernel Module **(LKM)**
* Create and Install a simple Windows device driver

Reading

* Slides Module 6 -LKM.
* <https://docs.kali.org/development/recompiling-the-kali-linux-kernel>
* <https://www.cyberciti.biz/tips/compiling-linux-kernel-26.html>
* <https://www.kernel.org/doc/html/latest/kbuild/makefiles.html>
* <https://linux-kernel-labs.github.io/master/labs/kernel_modules.html>
* <https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/writing-a-very-small-kmdf--driver>

Introduction

The simplest way to introduce code into a running kernel is through a loadable kernel module (LKM), which is a kernel subsystem that can be loaded and unloaded after boot-up, allowing a system administrator to dynamically add and remove functionality from a live system. This makes LKMs an ideal platform for kernel-mode rootkits. In fact, the vast majority of modern rootkits are simply LKMs. Most of device drivers are implemented as modules that can be loaded or unloaded from memory at runtime without recompiling the entire kernel. Such drivers are still part of the kernel but compiled separately and enable only when loaded. Rootkits can be written to hack existing device drivers or a new device driver can be written to hack the system.

Lab Requirements:

To complete this Lab you need the following:

1. Virtual-Box latest version
2. Linux virtual machine
3. Windows virtual machine

1.0 Analyze and create Linux LKM \_\_\_/20

The objectives of this exercise is:

1. Explore Linux LKMs
2. Create and insert simple Loadable Kernel Modules -LKMs
3. Modify created LKM
4. **( 1 marks )**Kernels can be recompiled to customize kernel features required by specific hardware or software. In order to compile the kernel or install kernel modules the **make utility** is required. Use Linux manual to learn the main purpose of **make** utility. What is the purpose of make utility?

**The make utility will determine automatically which pieces of a large program need to be recompiled and issue the commands to recompile them.**

1. **(1 mark )**Explore **/boot** directory to identify kernel image **vmlinuz** and respective release. You can also use the command **uname –r** to verify the current kernel release. What is the kernel release of your system?

**5.13.0-35-generic**

1. **(1 marks)** The kernel source tree is the best reference and learning tool for kernel hackers. Explore Linux Kali2020 kernel source directory **/usr/src/linux-headers-5.7.0-kali1-common.** Display Makefile content: **cat Makefile** and analyze the rules, parameters and targets of this file.

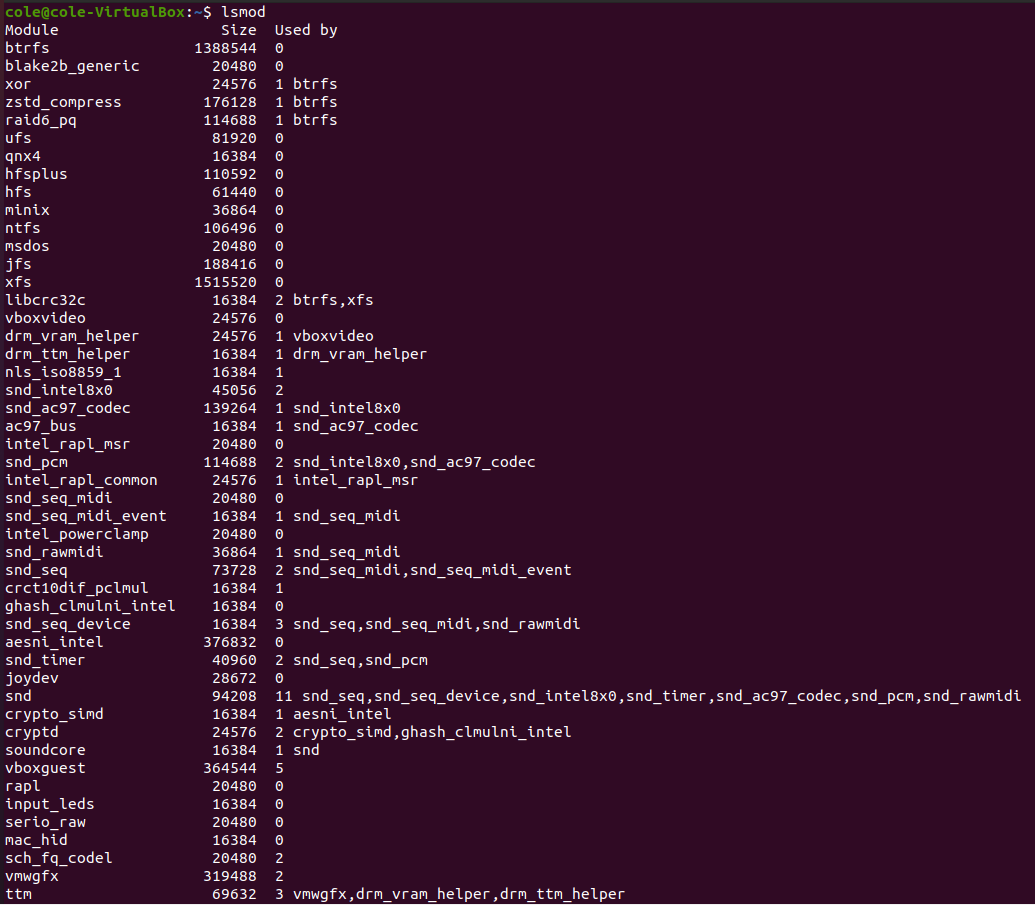
Makefile can be very complex. To learn makefile intro access the following web site: <https://www.gnu.org/software/make/manual/make.html#Introduction> . Access the following site: <https://www.gnu.org/software/make/manual/make.html#Phony-Targets> and explain the purpose of .PHONY target

**.PHONY is used to specify that the target is not a file**

1. **(1 marks)** Use linux manual to learn about lsmod, modprobe and insmod commands. What is the difference between insmod and modprobe?

**Modprobe allows you to add and remove modules from linux kernel, insmod only allows you to add.**

1. **(2 marks)** Use lsmod to verify current loaded kernel modules and attach the screen capture that displays current LKM loaded in this Linux released

****

A Loadable Kernel Module (LKM) is a compiled object file that you can load into kernel space, it becomes part of the running kernel to provide a new functionality

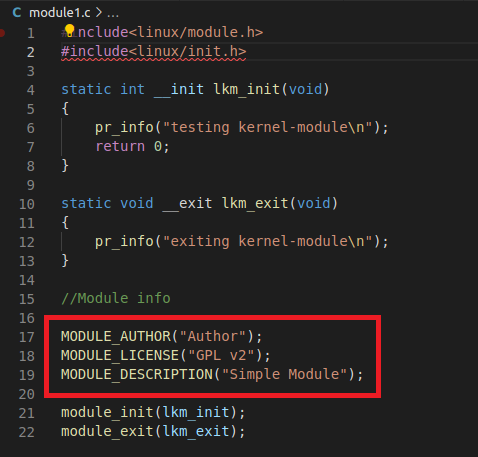
* 1. Use your favorite text editor to crate the following LKM save it as module1.c



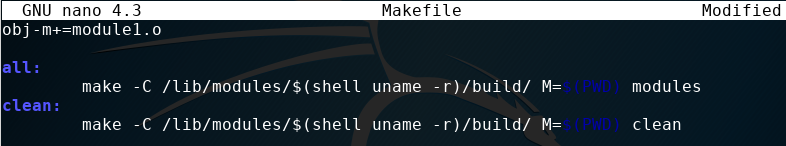
This module uses pr\_info( ) to print when module is initialize and when exists

This module will be statically loaded as part of the kernel. \_\_init initializes the function and is removed right after initialization, releasing memory for other functions. \_\_exit is used to clean up.

* 1. (3 marks) Attach screen capture of module1.c and identify 3 components of kernel modules?



* 1. After the module is created it has to be compiled as a kernel module with the *make* utility which in turns requires a Makefile. The Makefile required to compile the kernel is more complex than Makefile required to compile a kernel module. Create the following Makefile and use TAB for indentation.

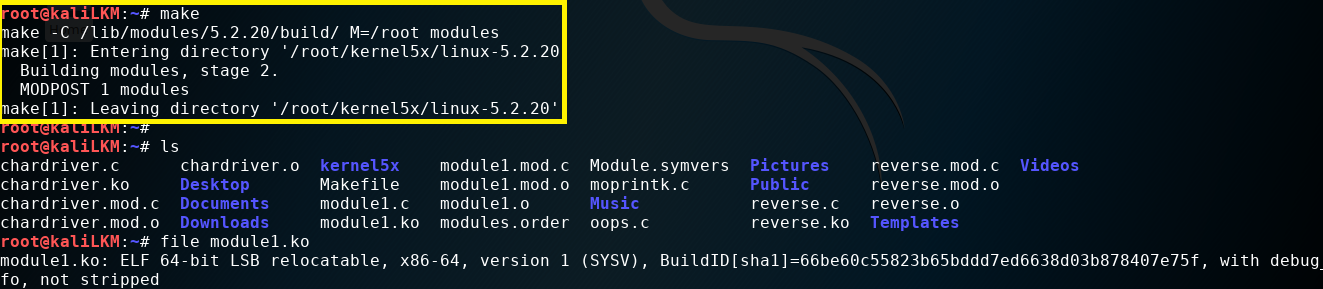


make utility builds object files. The first line in Makefile contains the object file name. In this case if the module is module1.c then the object should be module1.o The second line calls make utility and change to the directory (path) that contains kernel source /lib/modules$(shell uname –r) and compile the modules located in current directory $(PWD). The third line is to undo make when issuing make clean.

NOTE: Paid attention to the syntax M=$(PWD) modules. In some files $PWD is not visible because is blue color in the screen capture. If you missed $PWD the Makefile will generate an error

* 1. Now module and Makefile are ready you can compile the module

#make



* 1. After compiling the module use *ls* command and identify the file generated with extension .ko in this case should be module1.ko
  2. ( 2 marks) Attach screen capture with Makefile and results after compiling the module
  3. Use file module1.ko to verify module format
  4. Use modinfo module1.ko to verify module information or settings
  5. Open a second terminal to monitor the kernel log as follows:

tail -f /var/log/kern.log

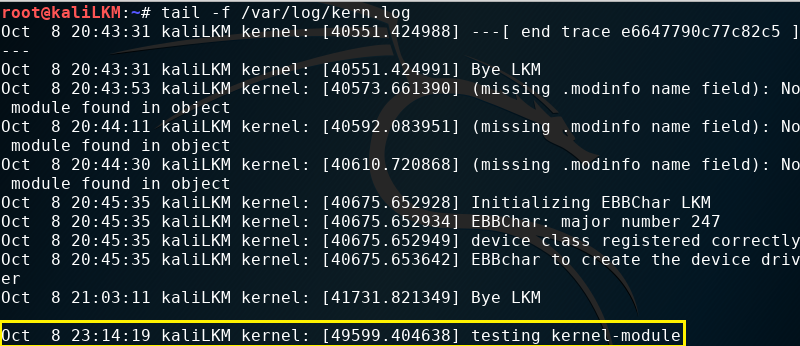
You can verify the log when the module is inserted or removed

* 1. On the terminal where you created module and Makefile use:

insmod module1.ko to insert the module

lsmod | more to verify if the module was inserted

* 1. Check the kern.log for initialization messages on second terminal and analyze the results



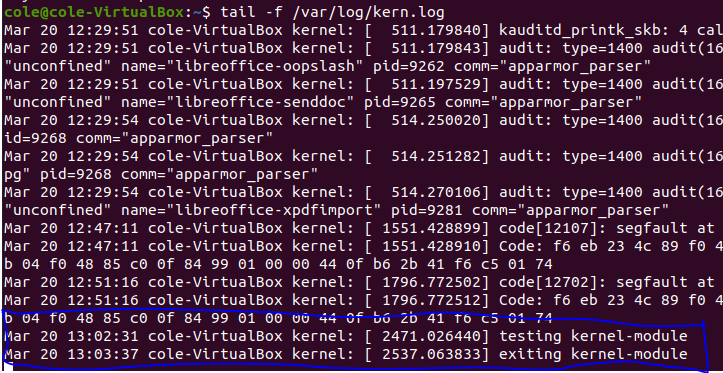
* 1. You can remove the module with rmmod module1 check the messages from kern.log on the second terminal and use lsmod to verify if the module was removed.
  2. Use *make clean* to clean up. Use *ls* to verify the results. All .o and .ko files should be removed.
  3. (3 marks) Attach screen capture that demo the used of respective commands to
     1. Insert the module in the kernel



* + 1. Verify module information



* + 1. Kernel.log with results that demo inserted module



* 1. (1 mark) Access the following web site: <https://www.kernel.org/doc/htmldocs/kernel-hacking/common-routines.html> to learn about printk( ) function. What header is required by this function?

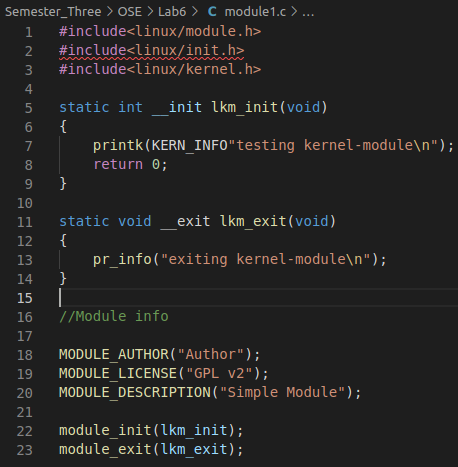
include/linux/kernel.h

* 1. Modify the current module1.c by replacing the function pr\_info( ) with printk( KERN\_INFO “ testing new function \n”);

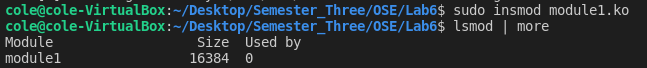
Pro-tip: KERN\_ALERT is the priority of the output message. There are many

others such as KERN\_INFO.

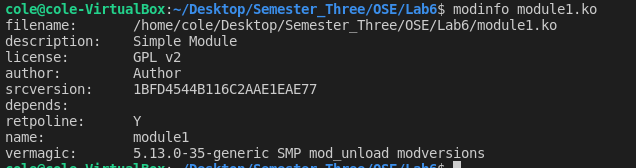
* 1. Save the module as moprintk.c
  2. Modify Makefile with the respective object file. Every time you want to compile a new module Makefile has to be modified with respective object file name
  3. Compile the new module
  4. Insert the module and monitor the results with kern.log
  5. ( 5 marks ) Attach screen capture with modified module and the used of respective commands that demo:
     1. Modified and compiled LKM



* + 1. Inserted module in the kernel



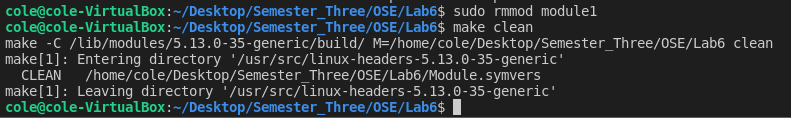
* + 1. module information



* + 1. based memory address (offset) of the module inserted



* + 1. how to remove the module



2.0 Character Device Driver Module –LKM \_\_\_/23

The objective of this activity is:

* 1. Create a more complex character device driver module
  2. Download a character device module, analyze the module and insert it into the kernel

In Linux devices are classified into block (with file system) and character devices (raw devices that transmit data character- by character. Device driver can be used to pass information between user-space and LKM in kernel space. Linux devices have major and minor number to identify or associate the device driver. The **major number** is used by the kernel to identify the correct device driver when device is accessed and **minor number** is a device dependent and handle internally within the driver for each device instance.

Rootkits can be written to hack existing device drivers or a new device driver can be written to hack the system.

1. **(3 marks)** Change to dev directory: **cd /dev** and **ls –l** to explore this directory. Identify tty device and all instances of this device tty0, tty1…. The **c** in the first column represents a character device, the number **4** is the major number for tty. The kernel uses this number **4** to identify the driver for terminals (tty) and the numbers in the next column are the minor number, specific for each instance of this device. Explore /dev directory and identify the major and minor number of the following character device:

a. vboxguest

b. console

c. zero

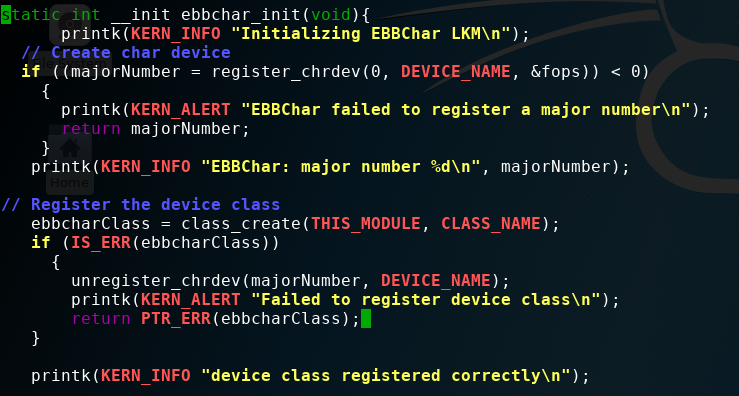
**A. Create a complex character device driver module**

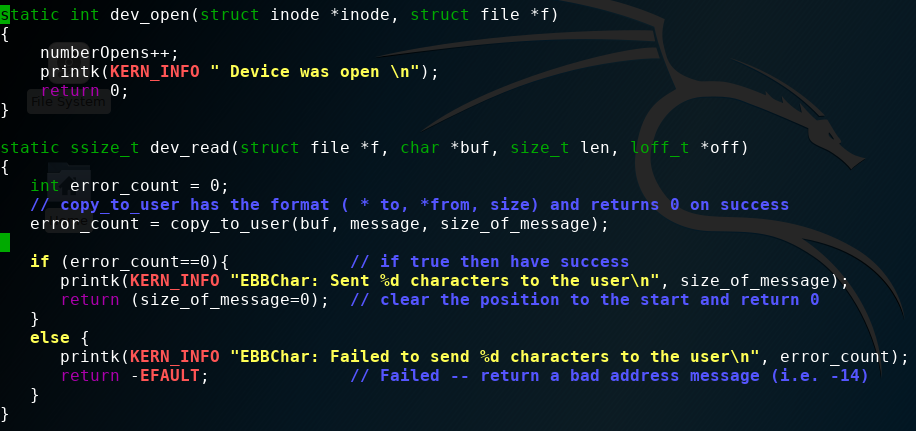
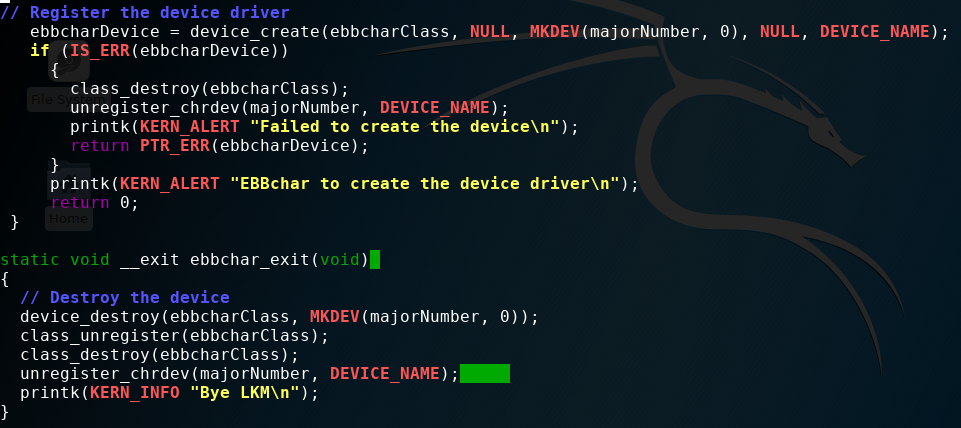
1. Now create the following character device driver module and analyze the main components of the module. Similar to previous modules it has **init()** and **exit()** functions but we need file operations required for character device and I/O such as:

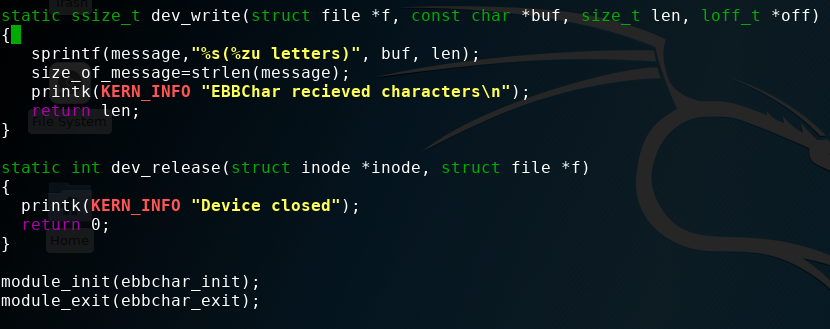
Open(), read(), write() and release(). In the module under //Prototypes analyze the arguments of these functions. Drivers have a **class name** and **device name.**





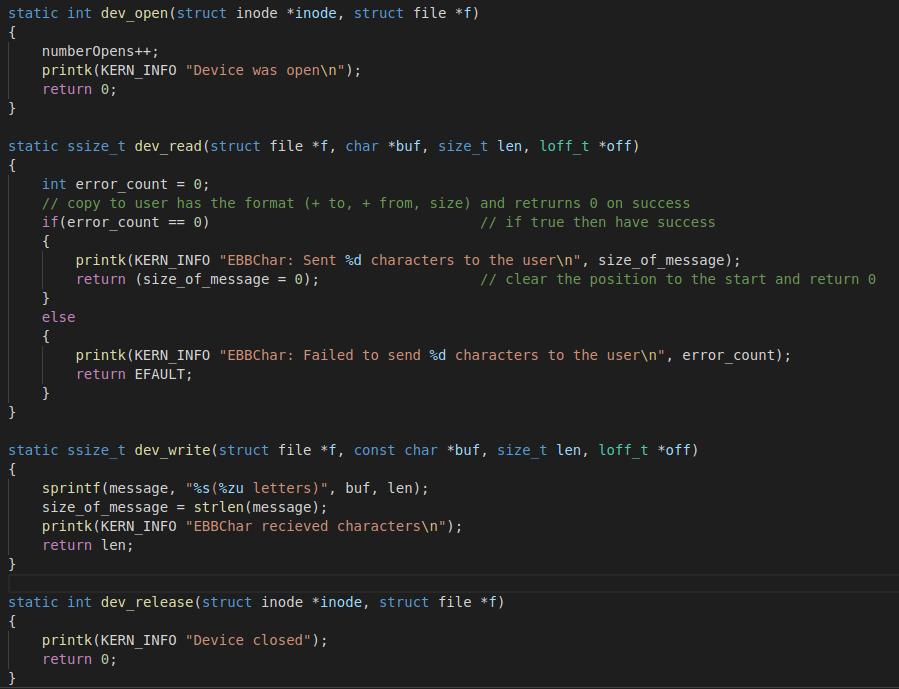






3. Save the module as chardriver.c

4**. (5 marks)** attach screen capture of modules chardriver.c created. Analyze the module components and answer the following question:



* 1. What is the driver’s device name and class name?

**#define DEVICE\_NAME "ebbchar"**

**#define CLASS\_NAME "ebb"**

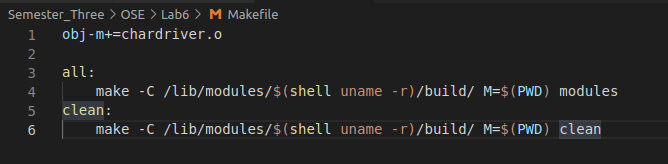
* 1. Analyze ebbchar\_init( ) function. What is the purpose of this function in this module?

**To initialize the driver and register the device class and device driver.**

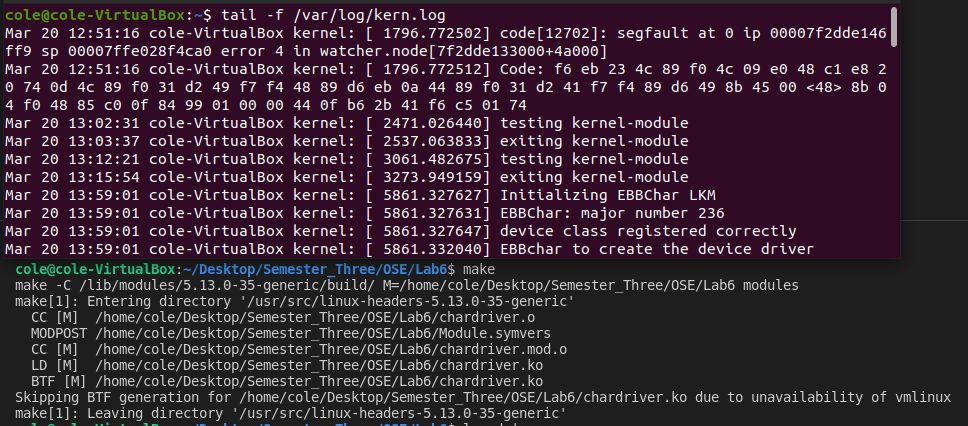
* 1. Analyze dev-open, dev-red and dev-write functions. What is the purpose of dev-write() function?

**It copies an entered string into the message string variable.**

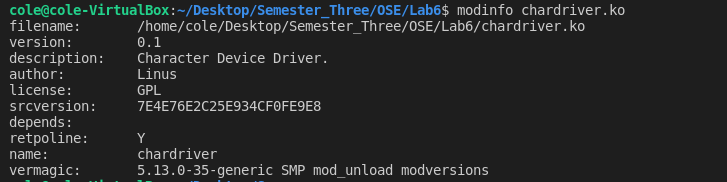
1. Modify Makefile with the respective object file.
2. Compile the character device driver module
3. Insert the module chardriver.ko and monitor the results with kern.log
4. Use lsmod to verify if the module was inserted.
5. Use cat /proc/modules to find the base memory address of the module
6. On a second terminal change to /dev directory and identify the new character device. Use ls –l to display the major and minor number of the device.
7. (5 marks) Attach the screen captures that demo:
8. Modify Makefile with the respective object file.



1. Compiled module and kern.log that demo module inserted



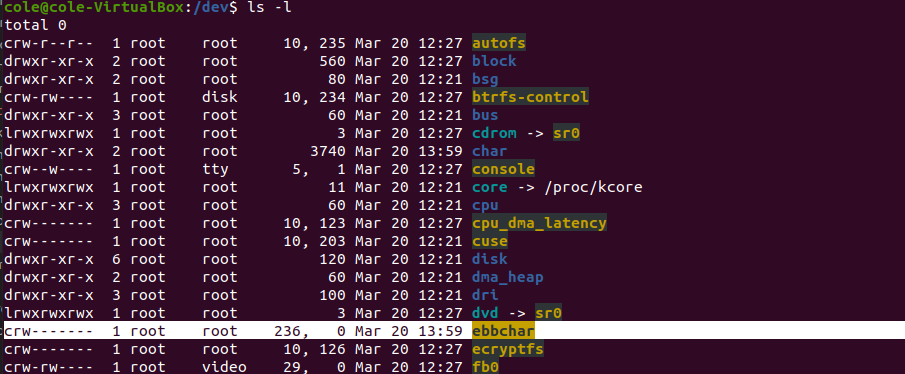
1. Module information.



1. The base memory address of the module



1. New character device inserted in /dev and its respective major and minor numbers

z

**B. Download a character device driver module and insert the module into the kernel**

Now that you created a complex character device driver module we can analyze a more sophisticated module

1. **( 2 marks)** Download the following **reverse.c** module from <https://github.com/vsinitsyn/reverse/blob/master/reverse.c> .The module is explained at <https://www.linuxvoice.com/be-a-kernel-hacker/>

2. **( 3 marks)** Analyze the module and answer the questions. In this module you will see similar functions as the one you created before. File operations are required open(), read(), write() for character devices. It also have typical init() function that registers the device and exit() function that unregisters the device . One difference here is **mutex lock**. Identify the mutex in the module.

a. Why are mutex required?

b. What is the buffer size?

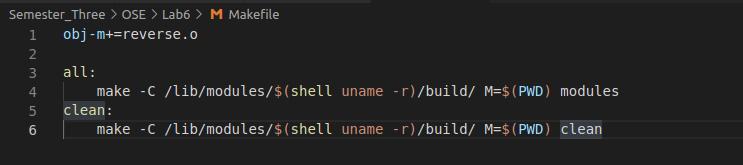
c. What is the purpose of this device?

3. Modify Makefile respectively and compile the module

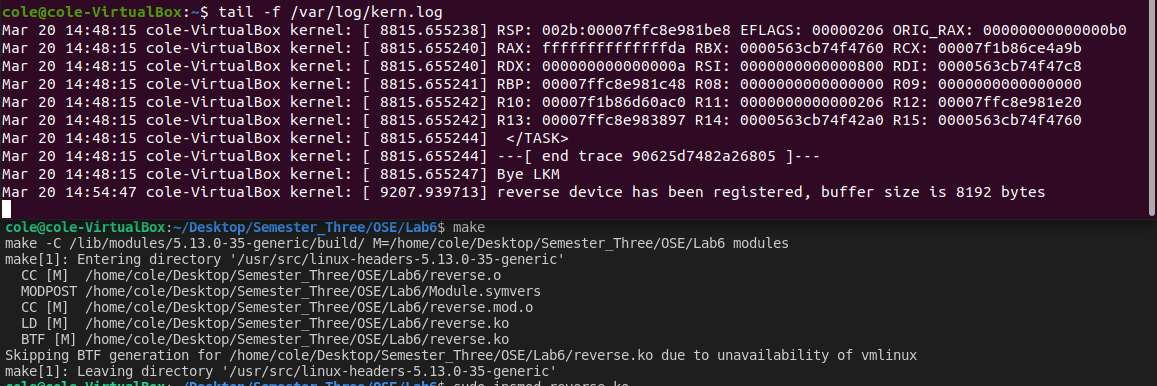
5. Use respective tools or commands to demo inserted module

6. **(5 marks)** Attach the screen captures that demo:

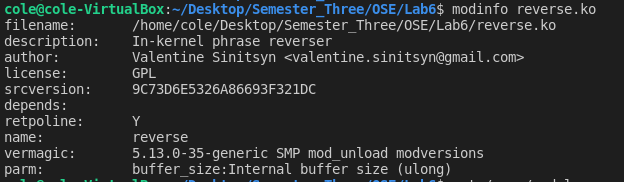
* 1. Modify Makefile with the respective object file.



* 1. Compiled module and kern.log that demo module inserted



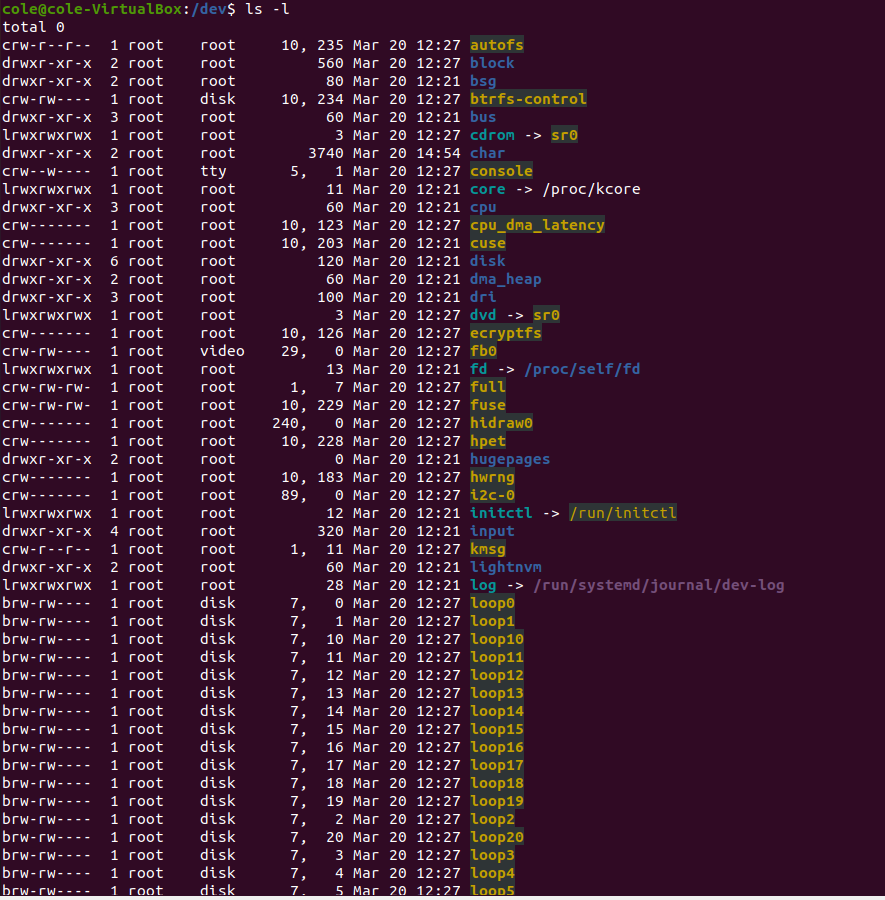
* 1. Module information.



* 1. The base memory address of the module



* 1. New character device inserted in /dev and its respective major and minor numbers

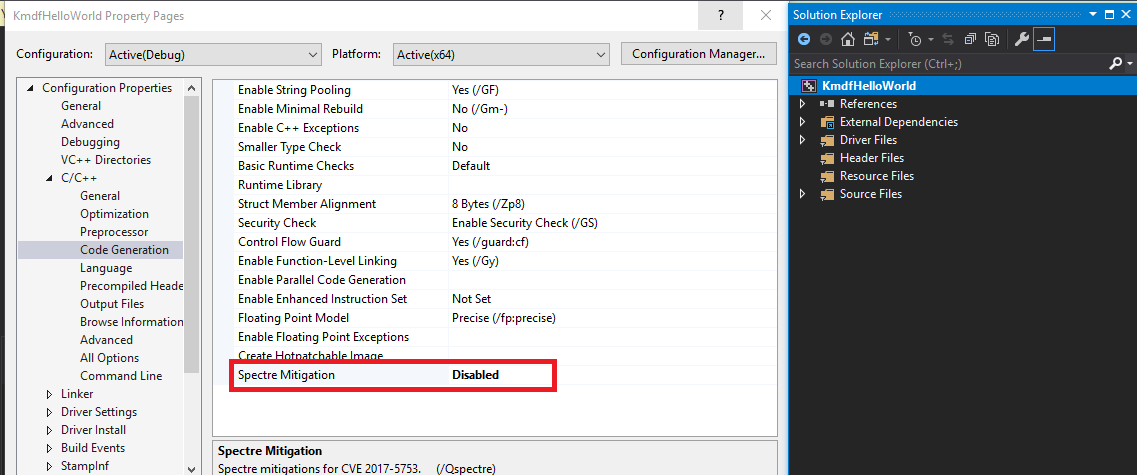


* 1. Windows Device Driver \_\_\_\_/10
* The objective of this activity is to write a simple Windows kernel driver

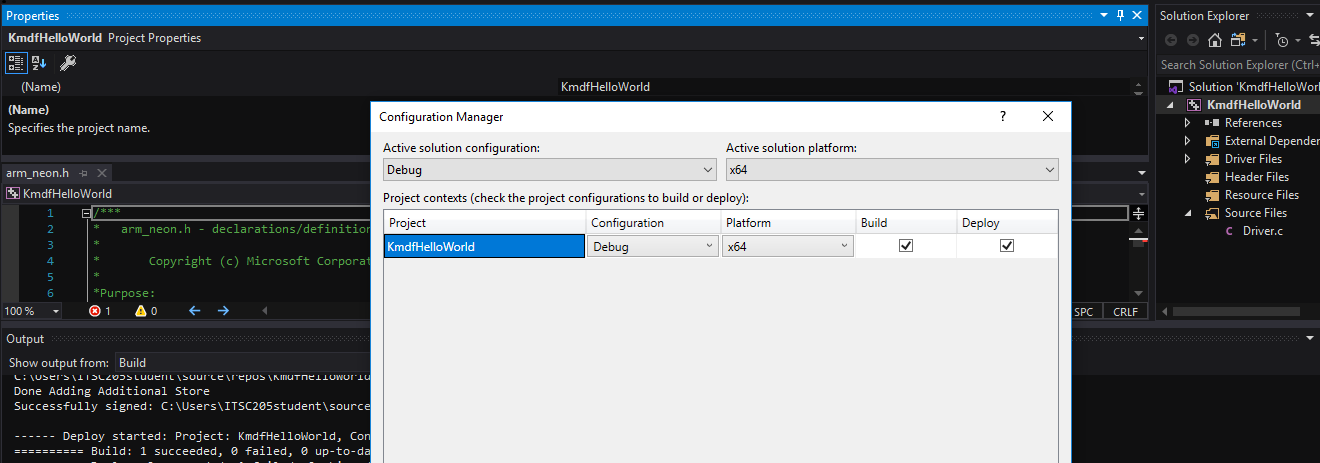
One of the latest Windows rootkits “ Detrahere” “ Return of kernel rootkit in Windows 10 “ from <https://www.slideshare.net/MSbluehat/bhv18-return-of-the-kernel-rootkit-malware-on-windows-10> installs a kernel driver that loads additional payload drivers from the hidden file system. In order to understand kernel drivers is a good practice to write a simple one.

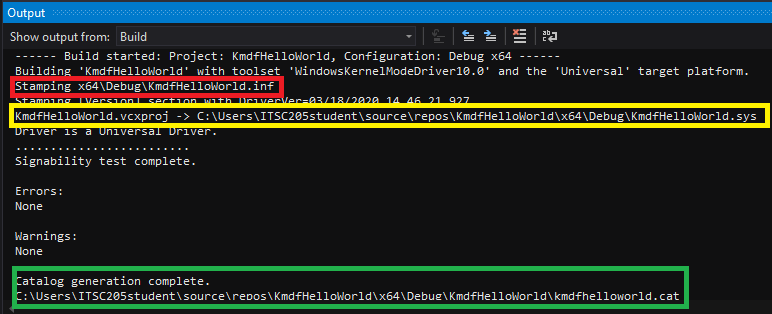
# Use Windows 10 with Microsoft Visual studio and Windows SDK installed. To write a universal Hello World Driver (KMDF) follow Microsoft instructions from <https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/writing-a-very-small-kmdf--driver> Write and analyze the code. Save it .You do not have to deploy it.

1. In order to create a Windows driver you need to have Microsoft Visual Studio and Windows Driver Kit. You can install it from <https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk>. Microsoft Visual Studio Community 2019 requires minimum Windows 10 release (build) 15063. It will not work with previous builds
2. WDK is a small program that will install the extensions and enables the development of device drivers on Visual Studio. Once you install it you can select under “ Create a New Project” the “Empty KMDF Driver” template that includes build environments and tools for driver developers.
3. Once you create the project you can create and build the driver by following this instructions. <https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/writing-a-very-small-kmdf--driver>
4. **( 2 marks)** Start Microsoft Visual Studio as Administrator. Copy or type the code as administrator. It requires high privileges. In the code pay attention and identify:
   1. Headers required
   2. Routine or function that initializes the driver once is loaded
   3. Functions that create device and driver objects.
   4. Macro that sends message to kernel debugger
5. Make sure you select Debug x64
6. Once the driver is built, it will create :
   1. .sys file which is the kernel mode driver
   2. .inf file that contains info about the driver
   3. .cat file catalog use to verify driver’s signature
7. The debugger will generate signature errors when running the program without administrator rights
8. If you get Spectre Mitigation error. Right click on KmdfHelloWorld , scroll down and at the end select Properties. It will open KmdfHelloWorld Property Pages. Select code generation and disable Spectre Mitigation:



1. Once device driver is compiled and debugged. Build will display the driver file with extension **.sys** , the **.inf** and **.cat** files respectively.





1. Now that the driver is created **KmdfHelloWorld.sys** you can install it by using the **sc.exe** tool. Do the following to install and load the driver
2. Open cmd as administrator and type:

**sc create hello type= kernel binPath= c:\Users\ITSC205student\source\repos\KmdfHelloWorld\x64\Debug\KmdfHelloWorld.sys**

NOTE: there is space between type= and kernel and between binPath= and the path. Also the path is the full path where the **.sys** file (which is the driver) is located. Make sure you have the right path

1. To test the installation, check the following registry key to find the installed driver

HKLM\System\CurrentControlSet\Services\hello

1. To start the driver you can use the sc.exe command. 64 bit systems drivers must be signed. Since this driver is no signed the only way to start it is to put the system in test signing mode and disable secure mode as follows:
2. Start cmd as administrator and type: **bcdedit /set testsigning on**
3. Make sure secure boot is disabled
4. Now you can start the driver as follows**: sc start hello**
5. To verify if the driver is running, open process Explorer tool from System Internals and search for the driver. You can click on search button and search either for hello or driver file **KmdfHelloWorld.sys**
6. **( 10 Marks)** Attach screen captures that demo:
   1. Downloaded and compiled code
   2. Files created once the driver is built .sys .inf and .cat
   3. Installed driver
   4. Driver running using sc command and process explorer

Note: Here is few basic kernel library functions that might be useful for Linux LKM. Refer to "man" pages for a detailed information.

|  |  |
| --- | --- |
| printk() | NAME  printk - print messages to console log  SYNOPSIS  #include <linux/kernel.h>  int printk(const char\*fmt, ...) |
| kmalloc(),  kfree() | NAME  kmalloc, kfree - allocate and free pieces of memory  SYNOPSIS  #include <linux/malloc.h>  void \* kmalloc (size\_t size, int priority);  void kfree (void \* \_\_ptr); |
| cli(),  sti() | NAME  cli, sti - disable/enable interrupts  SYNOPSIS  #include <asm/system.h>  extern void cli()  extern void sti() |
| get\_user(),  put\_user(),  copy\_from\_user(),  copy\_to\_user() | NAME  get\_user, put\_user, copy\_from\_user,  copy\_to\_user - copy data between kernel  space and user space  SYNOPSIS  #include <asm/uaccess.h>  err = get\_user ( x, addr );  err = put\_user ( x, addr );  bytes\_left = copy\_from\_user(void\*to,  const void \*from, unsigned long n );  bytes\_left = copy\_to\_user(void\*to,  const void \*from, unsigned long n ); |

2020, Southern Alberta Institute of Technology. All rights reserved.

This publication and materials herein are protected by applicable intellectual property laws.

Unauthorized reproduction and distribution of this publication in whole or part is prohibited.

For more information, contact:

Director, Centre for Instructional Technology and Development

Southern Alberta Institute of Technology

1301 16 Ave. N.W., Calgary, AB T2M 0L4