PA1 - Programming Assignment #1
CSCI 3753: Operating Systems
Josh Fermin And Louis Bouddhou

1. One advantage of having a loadable kernel module (LKM) instead of directly modifying the kernel would be to extend functionality on demand. In other words, if new functionality is required, instead of having to keep the functionality in the kernel (which wastes space), we can just add that functionality in a loadable kernel module. From this loadable kernel module we can add the functionality straight into the kernel without having to reboot the whole machine. If we had to modify the kernel directly to include the new functionality, we would have to reboot the whole machine in order to have access to the functionality.

2. To compile the LKM we have to set up a makefile as follows:

```
	obj-m += rkit.o

	KDIR = /usr/src/linux-headers-3.13.0-34-generic

	all:
		$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

	clean:
		rm -rf *.o *.ko *.mod *.sysmvers *.order
```

Once this is created, all we have to do is issue the command make in the working directory where rkit.c is and it will use then use the makefile to execute. This gives the following output:

```
	user@cu-cs-vm:~/Dropbox/OS/Assignments/Lab1$ make
	make -C /usr/src/linux-headers-3.13.0-34-generic
SUBDIRS=/home/user/Dropbox/OS/Assignments/Lab1 modules
	make[1]: Entering directory `/usr/src/linux-headers-3.13.0-34-generic'
	 CC [M]  /home/user/Dropbox/OS/Assignments/Lab1/rkit.o
	 Building modules, stage 2.
	 MODPOST 1 modules
	 CC      /home/user/Dropbox/OS/Assignments/Lab1/rkit.mod.o
	 LD [M]  /home/user/Dropbox/OS/Assignments/Lab1/rkit.ko
	make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-34-generic'
```

3.
a.) ssize_t is a type in which contains a valid byte size or a negative value which would tell you that there is an error. size_t is just to return a regular size in bytes.

b.) asmlinkage tells your compiler to look on the CPU stack for the function parameters, instead of registers. System calls are marked with asmlinkage tag, so instead of looking in registers for parameters, they look for it on the stack.

c.) copy_from_user copies a block of data from userspace to the kernel.

d.) The sys_call_table can be thought of as an “API” for the interactions between user space and kernel space. This contains a table of all the system calls’ handlers and their addresses.

To get around the not being able to reference the system table... First, we start with a range of addresses where we know the system call table will be. For 64 bit systems it is: #define START_CHECK 0xffffffff81000000 #define END_CHECK 0xffffffffa2000000. First we define a guess for the syscall table which will start at START_CHECK (this is a pointer). And then we use sys_close which we know is certain offset from the start of the table. This offset is __NR_close. We want the guess + the offset to match the sys_close. To do this we increment through the guess by one address each time until the address of sys_close matches our guess + the offset. Given that we know where sys_call in reference to the the system call table, we can easily find the starting address of the table through the offset.

e.) Malloc allocates a physically fragmented area of memory, while kmalloc allocates a physically contiguous memory chunk. This is useful because we want contiguous memory when we are searching through for the char *proc_protect, which is h1dd3n. We do not want fragmentation of the data. Also, kmalloc writes exclusively to the kernel whereas malloc is for userland.

f.) printk prints messages exclusively for the Linux Kernel. These messages are output to a file usually in /var/log/dmesg. Calling dmesg will will write kernel messages to standard output. It is important because it is always there and can be called from the interrupt or process content or while a lock is held. The following shows evidence of the printk:

```
[ 3809.784549] rkit: module license 'unspecified' taints kernel.
[ 3809.784555] Disabling lock debugging due to kernel taint
[ 3809.785957] rkit: sys_call_table is at: ffffffff81801400
```

g.) The first call to write_cr0, disables the write protect of the cr0 register, i.e. it allows superuser procedures to writing into read only pages. This allows us to write into the system call table. The second call to write_cr0 is called after we have written the hack of the function (hijack the table pointers) and turns the write protect back on.

h.) rkit_init first finds whether or not the system call table is there. If it is, it calls the find function and finds the starting address of the system call table. From there we disable the write protect of the cr0 register, which allows us to write to the system call table and hack it with a function. To pass data off and restore, we use the xchg() function and exchange the original pointer with the rkit_write function pointer (the details of the rkit_write function will be explained in part i). After writing this hack, we turn the write protection back on.

i.) rkit_write is a new function that will hack (replace) the old function. So instead of pointing to the old write function, we are now pointing to this new one. We first save the copy of the original write function to pass user space data off to kernel space (this is copy_from_user) so we can restore it when the module is unloaded via rkit_exit. Then we check if the write buffer (kbuff) contains the string “h1dd3n”. If it does free the allocated memory from kbuff and then return the file exist error, in other words, this function will tell that the given string “h1dd3n” does not exist. If you do not find the given string, then free the memory from kbuff and then use the userspace buffer that contains the original write system call, and return that data.

4.) It does not show h1dd3n in standard out because of rkit_write where it finds the string in the buffer and then returns EEXIST, which is an file exist error.

5.) lsmod output:

```
Module               Size          Used by
rkit                 12629         0
vboxsf               43786         0
snd_intel8x0         38153         2
snd_ac97_codec        130285       1
snd_intel8x0
```

The problem is that we can see our rootkit in the module list.

6.) echo h1dd3n works because of the xchg(&sys_call_table[__NR_write], (psize)o_write) function. In this function, we change the pointers back to where they were before, thus replacing the pointer to our hack (rkit_write) with the pointer to the original write system call.

7.) When we uncomment the first two lines:

```
list_del_init(&__this_module.list);
kobject_del(&THIS_MODULE->mkobj.kobj);
```

And install the rootkit (i.e. sudo insmod rkit.ko), we no longer see it appear in the list of modules when we issue the command lsmod.

8.) Since we cannot sudo rmmod rkit.ko, we must restart our machine to remove the rootkit. This is why it is better to comment this part out while writing the rootkit.