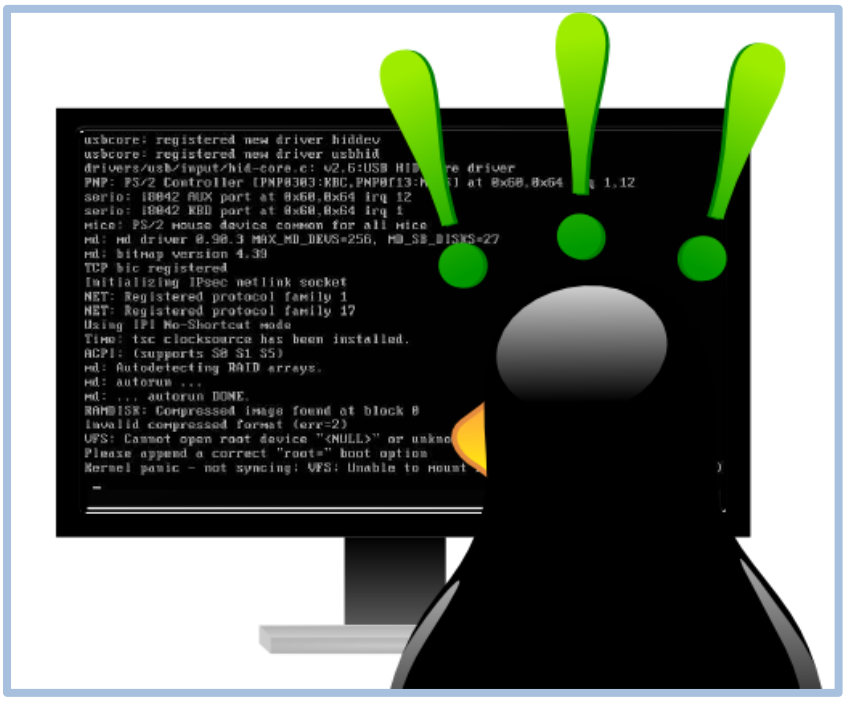
COMP3000: OPERATING SYSTEMS PROJECT REPORT



JOURNEY THROUGH THE KERNEL

AUTHOR: MUHAMMAD MUSTAFA (100823576)

DATE: 7th DECEMBER 2015

INTRODUCTION

Initially starting off my journey on to hacking the kernel, I had no idea where to begin and what to execute. The concrete purpose of the course being breaking the kernel, discovering patterns and implementing snippets of code or incorporate ideas to make the kernel do something that we would want it to was not something I was comfortable starting off with. I always have the habit of having a visual of the implementation or at least an idea to start off with, in order to work and understand code. It felt like jumping into the ocean not knowing how to swim. Regardless to that feeling, I started off my journey with baby steps into the kernel.

I began not having a purpose set in mind, which led to breaking the kernel in multiple ways e.g. a simple *printk()* statement in *fs/open.c* to see what one of the variables being used in one of the functions was doing. That broke my kernel. I had no clue as to why it did that. Later doing a *strace* on *ls* and checking */var/log/kern.log* I realized open.c is called multiple times and thus a lot of prints were being executed. The infinite loop to printing broke my kernel, setting a count and limiting the output took care of that problem.

Doing random experiments and printing variables was not going to get me anywhere on the journey to change the kernel. I needed a purpose and idea, to give me direction and narrow down my research to focus on a particular topic. Thus I came with an idea: If someone makes a folder named "Mustafa", doing ls on the folder should show all files named as "Mustafa" regardless of their original names.

DESIGN/IMPLEMENTATION:

ROUND 1:

To lead up to the idea, I had to figure out how the command ls reads the filename and prints it to bash, with that name. In that way if I know where it is stored and where the path to the created document is being made, I can definitely try and change the variables to the functionality I want it do.

To start off, I did *strace touch vi myfile.txt*. This command would make the file named myfile.txt and strace calls all the respective system calls for the making this file. Since vi creates the file if it doesn't exist and opens it as well, doing a touch will just create it and leave the rest of the system calls. Looking at the respective system calls, I tried skimming through the calls trying to find anywhere I could find the name of my file being passed into one of the system calls. I noticed the *write()* system call taking myfile as an argument. I noticed *file* is used in *fs/read_write.c* in the *write()*. I tried finding other in stances of my assumptions where my filename could be going, linking from one file to another.

Next I stumbled across Checked *write.c: Linux/drivers/tty/synclinkmp.c***.** Line 92 uses *tty*, so I was assuming it to be printing to bash. It has *filename* as well also an argument call **buff*. I wanted to see where to see what buff was storing. So did a print for buff.

ROUND 2:

I also printed *buf* in the *DEFINE3(write..)* in fs/read_write .c. Inserted printk in the function. When compiling it gave me an error that it was not permissible the declaration of *struct fd f*. The fix to that was declaring the structs on top the print statement. Later I checked for any results. There were a lot of prints for write, speed skimmed through them, noticed a trend around the 1000..something line that the buff builds up to a path of some sort. Inserted if statement for the print statement, that it only prints when it's /home. Compiling went fine for that condition, but the kernel crashed, which made no sense to why it that. I checked the console; it printed my statement but does not let me access the kernel, kind of stays in the loop of printing my statements. I rebuilt the kernel and the tried to print buff again, now I tried and look for a pattern, and restrict it in home folder if I could. It failed, it does not build up to a path, it usually has s, c or a "."Although later a closely looking into the print some sort of filenames present when you ls. But I don't know what to do, and how to manipulate it; which meant, I had to take another approach.

ROUND 3:

Around this time, my aim was to get a complete understanding of how the filename was being around in the kernel. How the path were being made to store the path of my directory or file. I tried many other ways to approach my goal, but I was reaching nowhere. All I was doing

was going around files and reading code, that lead to other files and structs. This approach was not leading anywhere.

Went for "**getdents**" I figured starting off with that in **fs/readdir.c**, I printed vales for names and crashed the kernel couple of times, also noticing anything I try finding related to filename or name, doesn't show up on /var/log/kern.log but instead when I reboot the console it shows up on the rebooting and prints the path for name...and the kernel just crashes!

This lead me to change my idea altogether. To something a bit simpler so could come up with a definite solution. I modified my idea to: If I make a file named "WEIRD" the kernel should prevent that and name the file "DONE" instead. Starting all over again with: *strace mkdir WEIRD* . This command created then folder named WEIRD. The mkdir took the name of my file. This was now my starting point of my research, and see if I could change it there in the mkdir system call. The mkdir system call exists in fs/namei.c:

```
3529 SYSCALL_DEFINE2(mkdir, const char __user *, pathname, umode_t,
mode)
3530 {
3531         return sys_mkdirat(AT_FDCWD, pathname, mode);
```

As it calls the mkdirat syscall:

```
3504 SYSCALL_DEFINE3(mkdirat, int, dfd, const char __user *,
pathname, umode_t, mode)
3505 {
3506         struct dentry *dentry;
3507         struct path path;
```

My next approach was going via *mkdir in fs/namei.c*. the mkdir calls on *mkdirat()*..which in return takes *pathname* as its argument. pathname being of *struct path*, has a *dentry struct*, which in return has a *struct qstr* which holds a *cost unsigned char *name*. Reaching here I realized this might definitely be the name. Although for some reason it has happened to me more than once now, any changes I tried making in the kernel said: "**write error (file system full?)**", which I have no clue why it happens, but rebuilding the kernel was apparently the only way around.

My next approach was to print pathname in mkdir itself, just to test what pathname was. Turned out pathname was exactly the name I had been looking for. When I tried editing pathname using the one in the mkdirat, I was unsuccessful in doing so. I tried different ways, but the kernel would compile fine, and when creating the folder, I would get a message saying bad address. I did:

```
If(strcmp(pathname, "WEIRD") == 0) {
        char ss[] = "DONE";
        strcpy(pathname, ss);
}
```

Since this approach on changing the name didn't work, I did another strace on mkdir and noticed that the execve system call takes in arguments out which one was my filename.

```
student@comp3000:~/linux-source-3.19.0$ strace mkdir WEIRD
execve("/bin/mkdir", ["mkdir", "WEIRD"], [/* 20 vars */]) = 0
```

This led me in to the system call for execve, in fs/exec.c, where the fuction for the system call was as follows:

```
1665 SYSCALL_DEFINE3(execve,
1666                 const char __user *, filename,
1667                 const char __user *const __user *, argv,
1668                 const char __user *const __user *, envp)
1669 {
1670         return do_execve(getname(filename), argv, envp);
1671 }
```

Here I printed the argument argv, and noticed that for my mkdir command they were assigned argv respectively. In my case argv[0] was "mkdir" and argv[1] was "WEIRD". This was my break through! I changed the execve function to have:

```
1665 SYSCALL_DEFINE3(execve,
1666                 const char __user *, filename,
1667                 const char __user *const __user *, argv,
1668                 const char __user *const __user *, envp)
1669 {
```

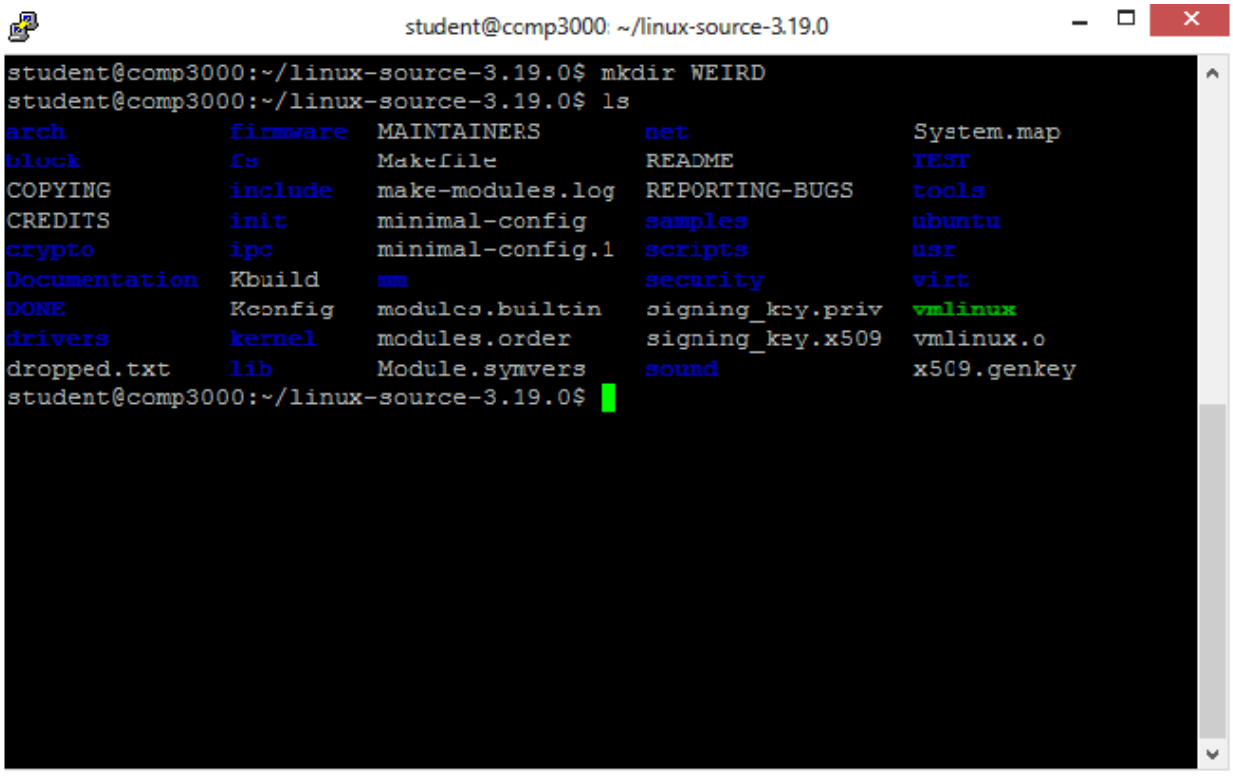
```
if ((strcmp(argv[0],"mkdir") == 0 && strcmp(argv[1],"WEIRD")==0)||(strcmp(argv[0],"vi") == 0 && strcmp(argv[1],"WEIRD")==0)){

char word [] = "DONE";
strcpy(argv[1], word);

}
```

Doing this, I change the argv[1] pointer to the desired name as argv is a static char *argv[]; The only way would be changing the pointer in the array of char, as the strings in it are immutable.

Now whenever someone does vi WEIRD or mkdir WEIRD, the kernel saves the name as DONE. It can be seen on doing ls. SUCCESS!!!

When I run:

mkdir WEIRD and the "ls" the file made is DONE. The same happens for when I do it with vi.



CONCLUSION:

There was a bit of contributions and assistance from fellow student who were working on their projects, which to completion of mine. The idea of understanding how the kernel works was tough altogether; the structs were something I struggled with also the pointers. I learned the copying of chars using strcpy. Then also const char* can be changed but the contents are immutable.

REFERENCES:

http://lxr.free-electrons.com/

rest of the work was experimenting and discovering on my own.