I put my testing all over the report, so the blue text describes testing/known bugs.

To begin this project, I realized that the classification folder must be in the userspace (a.k.a. /home/reptilian/… (for the most part how I used it). I went to /usr/ originally, but realized my error when I had permissions denied, and so the folder was put in the directory /home/reptilian. From there, I knew that I needed classification.h, classification.c (where the syscalls are invoked respectively for get & set for processes' classification), and a Makefile (with the appropriate compilation commands).

It is important to note that with only these three files (.c, .h, & Makefile), the classification.o and classification.a files would be autogenerated with "make". It is also essential to understand that the two library functions explained in the specification document – particularly set_classification and get_classification – are actually implemented within the kernel, and the classification.c file in the user space actually tests out both of those syscalls. Going further, I made the logic of implementation of the syscalls within the kernel itself, as connected through inspiration from xattr and functions that get/set respectfully.

Aside from that, since the library functions were to be in the kernel mode, I navigated to /usr/rep/src/reptilian-kernel and made my new directory called "el_proyecto_3." For the Makefile, I had to be confident that the kernel src had my new dir & I had to compile the one source file I made (fortunately, only one .c needed); the .c consists of calling the syscalls to get & set processes' classifications via the two library functions. To accomplish this feat, later on, I had to include my changes in the kernel's Makefile, as well. Navigating over to there, I made sure that it included my new directory (a.k.a. letting the compiler know my new folder holds new syscalls to be maneuvered). Moreover, within the .c file in my kernel folder, I had to find a manner in which every file is attached with the new classification. After looking through xattr.c and others, I used vfs_setxattr() as the vfs is logically an important place to make our classification appear. After that, I use kern_path respectfully to get the path struct, and, once that is received, I can progress to the next step, which is to find the dentry to know the precise file in which classification should be attached; this part proved very tricky for me, as StackOverFlow suggested modifying other files (and led to my kernel crashing). It took a long time of testing with crashing and rebooting (once even deleting a snapshot that just messed up) to find the right file to modify. I recreated my p3.diff several times due to this problem. Once that was solved, I was able to pass results from that of dentry via vfs and then the corresponding title & content of attribute, as this data is essential in going through the setter method style or the "bringing to life" of the attribute. We can change the classification from 2 to 10, for example. It is important to take into the account the usage of kmalloc to give sufficient storage in vfs_getxattr() when the get accessor style is gone through. I spent quite a while going through tests through the setting and getting of the classification. (By looking at kern_path results, it is no doubt possible to make a decision on file validity in a binary format, such that = 0 we know that we have reached invalid file, unfortunately). Continuing on, different processes have different things to see or do. Vfs indeed gives off certain codes (some are of blunders or err) and I looked at these to make decisions (-61 ==> fortunately, the file has set. Congrats!. -13 ===> sorry, you are not allowed). A "-1" is sent back to someone who is trying to access something they can't. Vfs_setxattr() is useful in making classification value 0 for that of file without a classification at all, as this was a case specified in the project preconditions. At first, it did not work, as my code was faulty. Most of the issues lied in the appropriate number of comparisons to make the valid conclusions & an ANDROID x86_64 # stuck screen (rarely). This appropriately handles permissions & makes things consistent and sensible.

In addition, I had to add my new syscalls to the kernel's understanding, and so I navigated into /include /linux/syscalls.h from the kernel & (using asmlinkage) made sure that my new syscalls' signatures were defined, making them understood and clear. I added my two lines at the bottom of the file (I used approach of asm linkages in el_proyecto_3.c). I also had to make my syscall's names and numbers defined, and navigating to /arch/x86/entry/syscalls/syscall_64.tbl proved to do the trick. I put my new syscalls under the existent table of x64 syscalls (from 0 to 331) & continued their trend. It was a colossal issue placing the new syscalls (directly affecting their numbers) & tables' spacing for declaring syscalls (the solution that allowed calls from classification.c to work was using solely tabs & no spaces in my other files). I was struggling to understand relationships between call numbers, entry vectors, and their format, too. Eventually, a blogger under a Linux Forum helped me. I broke my machine a couple of times over these small issues.
Furthermore, the project specifications state that "all files should be initialized with a classification of zero (0)" and, so I instantiated the classification attribute in include/linux/fs.h in kernel & made it zero in fs/inode.c. The goal was to put it in a place that ideally would make each new file have this classification attribute and then initialize it to 0. It proved to be a logical place to put the attribute as it fulfills the goal. This classifies that all files will have classification level of 0 and, if file classification not exist previously, the syscall created this.