Homework 4, Question 8

"I have no special talents.
I am only passionately curious."
— Albert Einstein

Many students seem to have extreme difficulties with this question. While it can be solved with just logical reasoning, this seems to me like learning swimming on dry land. Why not trying out what an actual UNIX system has to say? Seems obvious isn't it? ;o)

Environment

I know at least three ways of how to set up a testing environment without affecting my main computer, and which should work regardless of whether you have a Windows, MacOSX or Linux machine:

1. You can download Oracle's VirtualBox

   https://www.virtualbox.org

   There are binaries for Windows and MacOSX (I only tried out MacOSX). In addition, you need to download a Linux distribution. I used a recent iso-file of an Ubuntu distribution. All components are free.

2. If you happen to have a Raspberry Pi laying around (I have two for playing music as well as for all sorts of rainy-afternoon distractions). The cheapest model of a Raspberry Pi costs around £20. You also need an SD memory card of at least 4GB, which can be bought for £5 or less. Some SD cards come pre-installed with Linux, but all can be easily loaded with Linux. The good thing about Raspberry Pi's is that despite their miniature size and small cost, they are full-fledged Linux computers…exactly what is needed for such experiments. There are plenty Linux distributions on the Net that are tailored to work "out of the box" with Raspberry Pi's.

3. If you have a spare memory stick laying around, you can try out any of the live USB-versions of Linux.

   https://en.wikipedia.org/wiki/Live_USB

   The idea is to upload Linux on the USB stick, you plug it into your computer and boot up a Linux system without having to download anything to your computer. A notable live USB version of Linux is called Tails

   https://tails.boum.org

which comes with Tor pre-installed and is for people who need a maximum of privacy and anonymity (whistleblowers, dissidents). It is being said that journalists Laura Poitras and Glenn Greenwald used it when talking to Edward Snowden. Tails gives them anonymity even if their main system is compromised by malicious software, for example installed by the NSA.

However, a live USB Linux will need some support from the computer (BIOS) where you plug in the USB stick. I know Apple computers are a bit "special" with this and would need a 3rd-party boot loader for loading operating systems from an USB memory stick.

An alternative is to burn a CD/DVD with a live Linux distribution. But perhaps CDs/DVDs are already obsolete technology not available to everyone. The point is that loading an operating system from such a media is/was much better supported by various computers.

For my experiments below, I used option 2. In earlier versions of this module I have used option 1. I have not tried in a while option 3, but know that in the past I had a dedicated bootloader on an Apple computer just for the purpose of running operating systems from external disks. I also for a long time had spare CDs laying around just for the purpose that my (Linux) operating system got trashed enough so that it had to be rebooted externally.

Setup

Once you have Linux up and running, there are a few commands you need to know in order to replicate the ownerships and permissions from the question:

- `useradd` creates a new user
- `groupadd` creates a new group
- `adduser` adds a user to a group
- `chmod` changes the permissions of a file
- `chown`, `chgrp` change the ownership and group of a file

There is also a choice to be made what to use as microedit. If you do not want to make your hands dirty and write a test program yourself, I recommended to use the editors `vi` or `vim`, which is available on pretty much every UNIX system. For a first try out, this is a helpful choice for solving the question. However, it has a disadvantage: it will always assume you have read permissions to a file. To use these editors, I made a copy of them and renamed them to `microedit`. Be careful to set the setuid bit for `microedit`.

Permission Basics

The absolute basics is how the permissions are organised in essentially four blocks

This seems to be the knowledge everybody has. But already difficulties arise with the following fact, which could easily be resolved by a little experiment: assume a file is owned by Bob with permissions

```
-r--rw-rwx bob students file_name
```

The UNIX access rules imply that Bob will only have read access to this file, even if he is in the group students and the group access permissions allow read and write. Similarly every member in the students group who is not Bob, will only have read-write access permissions, not read-write-execute.

The question asked whether Ping, Bob and Emma can read or write the given files using the program microedit. This means we will call on the command line

```
> microedit file_name
```

for all files and for Bob, Ping and Emma. So if you want to find out whether Bob, say, can read or write a file, you need to find out what the access permissions with which `microedit` is run. This would be easy, if `microedit` did not have the setuid bit set. Then it would be just the rights of the caller (Ping, Bob or Emma). But your friendly lecturer arranged the question so that it has the setuid bit.

Recall that the setuid bit gives the program the ability to run with the permissions of the owner `microedit` file, not the permissions of the caller. I wrote in the handout

> *"The fundamental idea behind the setuid attribute is that a file will be able to run not with the callers access rights, but with the rights of the owner of the file."*

Something similar is written in the Wikipedia entry for setuid

`http://en.wikipedia.org/wiki/Setuid`

This implies for deciding whether *file* is readable or writable is not determined by the caller, but by the permissions with which `microedit` runs. As you might know already, and can also see in the Figure 1 shown later, any *file_name* given on the command line will be handed over to microedit as string. It is the "responsibility" of `microedit` what to do with it.

There is one caveat however: We need to find out first whether the caller (Bob, Ping or Emma) can actually run `microedit`—that is has execute permissions for `microedit`. Once `microedit` runs, it will assume the permissions of

the owner of `microedit`. The question is now whether these permissions are sufficient to read or write the file *file_name*. The hints so far should already be useful for answering the first three columns.

For the other two files we have to take into account that they are inside a directory. For directories apply special access rules. In the handout I wrote

> *"There are already some special rules for directories and links. If the execute attribute of a directory is* not *set, then one cannot change into the directory and one cannot access any file inside it. If the write attribute is* not *set, then one can change existing files (provide they are changeable), but one cannot create new files. If the read attribute is* not *set, one cannot search inside the directory (`ls -la` does not work) but one can access an existing file, provided one knows its name."*

With this also the last two columns can be filled in.

Advanced Permissions

While all hints so far should get you very close to the intended answers, there is one further complication arising from the setuid bit. The question asked:

> *…whether Ping, Bob, or Emma are able to obtain the right to read (R) or replace (W) its contents using the editor microedit.*

Note the underlined phrase. That means we need to ensure that there is no other way for Bob, Ping and Emma to obtain reading or writing permissions with `microedit`. Actually there is. Any file that has the setuid bit set will be called with the permissions of the owner, but once it has done the work, it can "lower" the permissions again to the callers rights. This is a second possibility we have to check whether the files become readable or writable when the permissions of the caller are re-instated. In the handout I wrote about the setuid-program `passwd`:

> *"As an example consider again the `passwd` program. When started by, say the user `foo`, it has at the beginning the identities:*
>
> - real identity*: `foo`*
> effective identity*: `foo`*
> saved identity*: `root`*
>
> *It is then allowed to change the effective identity to the saved identity to have*
>
> - real identity*: `foo`*
> effective identity*: `root`*
> saved identity*: `root`*

> *It can now read and write the file* `/etc/passwd`*. After finishing the job it is supposed to drop the effective identity back to* `foo`*. This is the responsibility of the programmers who wrote* `passwd`*. Notice that the effective identity is not automatically elevated to* `root`*, but the program itself must make this change. After it has done the work, the effective identity should go back to the real identity. "*

It was hoped by your friendly lecturer that any of the students would have consciously considered this possibility, but alas nobody did…

A Program in C

I suggested above to use a copy of the editors `vm` or `vim` for `microedit`. This works reasonably well, except for one instance: if a file is not readable, then these editors will not be helpful for checking whether the file is writable. Giving out such a permission is a perfectly "normal" situation in many large UNIX systems. A user might be allowed to write into central log files, but should not be able to read them (otherwise they can find out what other users did). To get around this problem, I brushed up my C knowledge from school days and googled around for how to read and write files. Typing in "read write in C" in the all-knowing search engine, I obtained the link

`https://www.cs.bu.edu/teaching/c/file-io/intro/`

which tells you pretty much everything what there is about opening a file in C for reading and writing. (There are certainly more and better sources for finding out how to read and write files. This was just at my finger tips.) A little bit more googling helped me to display the user that determines the access permissions. Being lazy, I did not spend a thought of refactoring the file to be as small as possible, and also did not go the extra mile to convert the ID of the user into a clear name.

The resulting little C program is shown in Figure 1. It explicitly checks for readability and writability of files. The `main` function is organised into two parts: the first checks readability and writability with the permissions according to a potential setuid bit, and the second (starting in Line 34) when the permissions are lowered to the caller. Note that this program has one problem as well: it only gives a reliable answer in cases a file is **not** readable or **not** writable. In these cases it returns an error code 13 (permission denied). It sometimes claims a file is not writable, say, but with an error code 26 (text file busy). This is unrelated to the permissions of the file.

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

FILE *f; //file pointer

//tests return errno = 13 for permission denied
void read_test(char *name)
{
    if ((f = fopen(name, "r")) == NULL) {
      printf("%s is not readable, errno = %d\n", name, errno);
    } else {
      printf("%s is readable\n", name); fclose(f);
    }
}

void write_test(char *name)
{
    if ((f = fopen(name, "r+")) == NULL) {
      printf("%s is not writable, errno = %d\n", name, errno);
    } else {
      printf("%s is writable\n", name); fclose(f);
    }
}

int main(int argc, char *argv[])
{
    printf("Real UID = %d\n", getuid());
    printf("Effective UID = %d\n", geteuid());

    read_test(argv[1]);
    write_test(argv[1]);

    //lowering the access rights to the caller
    if (setuid(getuid())) {
      printf("could not reset setuid, errno = %d\n", errno);
      return 1;
    }

    printf("Real UID = %d\n", getuid());
    printf("Effective UID = %d\n", geteuid());

    read_test(argv[1]);
    write_test(argv[1]);

    return 0;
}
```

Figure 1: A read/write test program in C. It returns errno = 13 in cases when permission is denied.