CS 465 Introduction to Computer Security
Programming Assignment #2

Assigned: Tuesday, November 12, 2013

Deadline for the electronic submission: 11:59 pm, Tuesday, December 3, 2013
Deadline for the hard copy: 10 am – 12 noon @ 721 ESB on Wednesday, December 4, 2013

NO LATE ASSIGNMENTS

The goal of this assignment is to implement a system which emulates a simplified version of the access control mechanisms used by UNIX and UNIX-like operating systems. The system consists of user accounts, groups of users, and files. There is a privileged user, called `root`, who is able to perform administrative tasks such as creating user accounts and groups or changing the owner of a file. Each user account has a username and password. A user may belong to zero or more groups. Each file has an owner, a triplet of access permissions, and an optional group attribute associated with it. As any trusted operating system, your system will log its activity in an audit log file.

The programming assignment may be implemented in Java, C++, Python, or C. The program must be compiled and run on the LCSEE Ubuntu Linux *shell* servers.

The program executable must be named `access`, or the closest possible equivalent depending on the programming language that you chose. For example if you use Java, the file name for the class containing the main method should be `access.class`. The program will be invoked by the following command line (or equivalent depending on programming language),

```
./access filename
```

where `filename` is a command line argument that stands for the name of an ASCII file containing the instructions that the program should interpret. An example of this could be:

```
./access testcase1.txt
```

The program must accept the name of a text file as input via a command line argument and respond appropriately based on a predetermined set of instructions in the file. Each instruction will be written on a separate line. It can be assumed that all instructions will be free from any syntax errors, that is, each line in the input file will contain a syntactically correct instruction.

The following is the list of instructions:

`useradd` *`username password`*

`login` *`username password`*

`logout`

`groupadd` *`groupname`*

`usergrp` *`username groupname`*

`mkfile` *`filename`*

`chmod` *`filename rwx rwx rwx`*

`chown` *`filename username`*

`chgrp` *`filename groupname`*

`read` *`filename`*

`write` *`filename text`*

`execute` *`filename`*

`ls` *`filename`*

`end`

In order to create the super user for the system, the very first line of the instructions file has to be of the form:

```
useradd root password
```

This will be the ONLY time when the `useradd` command can be executed without a user being logged in or by a user other than `root`. If the very first command is not of the form described above, then your program should report an appropriate error (both to the console and audit log) and terminate. After execution of this command, the `root` user will still need to login to perform any further actions. Once logged in, the `root` user will have administrative privileges and will be able to issue any valid command (except `login`) regardless of the access permissions.

A username, group name, or filename may consist of up to 30 ASCII characters except for a forward slash ('/'), colon (':'), or any of the following white space characters: carriage return, form feed, horizontal tab, new line, vertical tab, and space. A password may consist of up to 30 ASCII characters except for the following white space characters: carriage return, form feed, horizontal tab, new line, vertical tab, and space. Instructions, usernames, passwords, group names, and filenames are all case sensitive.

Each user account has a password. A list of users and corresponding passwords should be maintained in an ASCII text file called *accounts.txt*. You may organize this file any way that you like as long as it is used by your program for user authentication AND you clearly and thoroughly explain your approach and the format of your *accounts.txt* file in your program documentation.

A user may be a member of zero or more groups. Your program should use an in-memory data structure to keep track of user groups. After all instructions have been processed, but prior to terminating, your program must create a file named `groups.txt`. The `groups.txt` file should contain a listing of each group and users who belong to the group. The exact file structure will be provided as part of example test cases.

Each file has associated with it an owner, at most one group of users and a set of permissions. Your program should use an in-memory data structure to keep track of this file meta-data during execution. After all instructions have been processed, but prior to terminating, your program must create a file named `files.txt`. The `files.txt` file should contain a listing of each file name, with its owner, group, and permissions. The exact file structure will be provided as part of example test cases. For each file created with the command `mkfile`, your program must also create a physical text file and provide the ability to read from and write to the file. The file should be created in the same directory as the program executable (i.e. the "current" directory).

Upon the execution of each instruction, the program should DISPLAY an appropriate comment on the screen AND LOG the SAME comment into an audit log file called *audit.txt*.

Since the *accounts.txt, audit.txt*, *groups.txt*, and *files.txt* files serve a special purpose they should not be accessible to a user via the `mkfile`, `read`, `write`, `execute`, `ls`, `chown`, `chgrp`, or `chmod` commands. An appropriate error message should be displayed and logged if either of these file names are referenced using any command.

The user accounts, groups, files, and permissions do not need to persist across multiple executions of your program. That is, each execution of your program should start with a "clean slate". (The *accounts.txt, audit.txt*, *groups.txt*, and *files.txt* files should be completely overwritten if they already exist when your program first starts. This is also the case with any files that were created via the `mkfile` command. If a file already exists when your program starts, its content should be overwritten. However, subsequent calls to the `write` command that are issued during a single execution of your program should append to the appropriate file.)

Detailed description of commands

`useradd` *`username password`*

- Example: `useradd tom cs465rules`
- Creates a user with a specified username and password
- By default, the user should **NOT** be a member of any group
- Username and password are stored in an ASCII file called *accounts.txt* for later use (Your program has to reference this file for user authentication when executing the `login` instruction).
- Things to consider / constraints:
    - Two users with the same user name MAY NOT be created
    - This instruction may only be executed by the super user `root`. The only exception is the first instruction which creates the root account, as previously discussed.

`login` *`username password`*

- Example: `login tom cs465rules`
- Checks username and password against records in the *accounts.txt* file. If the username and password are correct, logs the user into the system. Otherwise, displays and logs an error message, then denies login.
- Things to consider / constraints:
    - If a user is currently logged in, another user cannot login. (This system does not support concurrent users.)
    - Once logged in, the user will be able to use the `read`, `write`, and `execute` instructions. The `read`, `write`, and `execute` access is allowed/not allowed according to the UNIX file access rights security model.
    - The user will also be able to use the `logout`, `mkfile`, `chmod`, and `chgrp` as specified later in this document.
    - There MUST be a user logged in to be able to use the `read`, `write`, `execute`, `logout`, `mkfile`, `chmod`, and `chgrp` commands.

`logout`

- Example: `logout`
- Logs out the currently logged user

`groupadd` *`groupname`*

- Example: `groupadd students`
- Creates a group to which users may be added. (Note: A group may be associated with a file using the `chgrp` instruction explained later in this document.)
- Your program should use a data structure to maintain a listing of all groups that have been created and the user(s) that belong to each group.
- Things to consider / constraints:
    - This command can only be executed by the `root` user.
    - A group is initially created without any users belonging to it.
    - Two groups with the same name MAY NOT be created
    - A group may not have the name `nil`. (`nil` is reserved for the cases when no group is associated with a file.)

`usergrp` *`username groupname`*

- Example: `usergrp tom students`
- Adds the specified user to the specified group
- Things to consider / constraints:
    - A user can be associated with multiple groups
    - If the specified user or group does not exist, then an appropriate error message should be displayed and logged.

`mkfile` *`filename`*

- Example: `mkfile newfile.txt`
- Creates an empty file with default permissions (`rw- --- ---`).
- Your program should use a data structure to keep track of the file name, owner, group, and permissions.
- Things to consider / constraints:
    - If the file already exists, then an appropriate error message should be displayed and logged.
    - A user must be logged in to execute this command
    - The owner of the file should be the user who is currently logged into the system and issued the `mkfile` command
    - Initially no group is associated with the file (i.e., its group should be `nil`)
    - The default permissions for the file should be as follows: `rw- --- ---`

`chmod` *`filename rwx rwx rwx`*

- Example: `chmod file.txt rwx r-x ---`
- Changes the access permissions associated with a file
- Your program does not need to physically manipulate the file permissions, but it should maintain permission consistently via a data structure.
- The first `rwx` parameter indicates the permissions for the owner of the file. The second `rwx` parameter indicates the permissions for the group with which the file is associated. The third `rwx` parameter indicates the permissions for others.
- Permissions are specified via a three letter string consisting of the following characters: `r`, `w`, `x`, and `-`. The first letter of a permission string can be `r` or `-`. An `r` indicates that read access is granted, while an `-` indicates that read access is denied. The second letter of a permission string can be `w` or `-`. A `w` indicates that write access is granted, while an `-` indicates that write access is denied. The third letter of a permission string can be `x` or `-`. An `x` indicates that execute access is granted, while an `-` indicates that execute access is denied.
- Things to consider / constraints:
    - A user must be logged in to execute this command
    - This command can be executed by the current owner of file or by the `root` user.
    - If the specified file does not exist then an appropriate error message should be displayed and logged.

`chown` *`filename username`*

- Example: `chown file.txt tom`
- Changes the owner of a file.
- Things to consider / constraints:
    - This command can only be executed by the `root` user.
    - If the specified file or user does not exist, then an appropriate error message should be displayed and logged.

`chgrp` *`filename groupname`*

- Example: `chgrp file.txt students`
- Changes the group associated with a file.
- Things to consider / constraints:
    - A user must be logged in to execute this command.
    - Only a single group can be associated with a file.
    - If you want to disassociate a file from all groups, then `nil` should be used for *`groupname`*.
    - This command can be executed by the current owner of the file or by the `root` user.
    - The group for a file can only be changed to `nil` or a group to which the owner belongs. For example, suppose there are three groups `A`, `B`, and `C`. Now suppose that user `X` owns file1.txt and belongs to groups `A` and `B`. User `X` would be able to issue the commands `chgrp file1.txt A`, and `chgrp file1.txt B`, but NOT `chgrp file1.txt C`. This restriction does not apply to the `root` user. The `root` user may change the group associated with a file to any valid group.
    - If the specified file or group does not exist, then an appropriate error message should be displayed and logged.

`read` *`filename`*

- Example: `read file.txt`
- Displays the contents of a file
- Access to this command is granted or denied according to the UNIX file access rights security model.
    - If the current user is the owner of the file and read access for owner has been granted, then the instruction may be carried out.
    - Else if the current user is not the owner, is a member of the same group that is associated with the file, and read access for group has been granted then the instruction may be carried out.
    - Else if the current user is not the owner, is not a member of the same group that is associated with the file, and read access has been granted for others, then the instruction may be carried out.
    - Otherwise access should be denied.
- Things to consider / constraints:
    - If the file does not exist, then an appropriate error message should be displayed and logged.
    - A user must be logged in to execute this command

`write` *`filename text`*

- Example: `write file.txt some text`
- Appends a line of text to a file.
- Access to this command is granted or denied according to the UNIX file access rights security model.
    - If the current user is the owner of the file and write access for owner has been granted, then the instruction may be carried out.
    - Else if the current user is not the owner, is a member of the same group that is associated with the file, and write access for group has been granted then the instruction may be carried out.
    - Else if the current user is not the owner, is not a member of the same group that is associated with the file, and write access has been granted for others, then the instruction may be carried out.
    - Otherwise access should be denied.
- Things to consider / constraints:
    - If the file does not exist, then an appropriate error message should be displayed and logged.
    - A user must be logged in to execute this command
    - The text should be APPENDED as a new line to the end of the file. Make sure your program does not totally overwrite the file.

`execute` *`filename`*

- Example: `execute prog`
- Emulates the execution of a file
- This should determine whether or not execution is permitted, then display an appropriate message such as "*prog executed successfully*".
- Access to this command is granted or denied according to the UNIX file access rights security model.
    - If the current user is the owner of the file and execute access for owner has been granted, then the instruction may be carried out.
    - Else if the current user is not the owner, is a member of the same group that is associated with the file, and execute access for group has been granted then the instruction may be carried out.
    - Else if the current user is not the owner, is not a member of the same group that is associated with the file, and execute access has been granted for others, then the instruction may be carried out.
    - Otherwise access should be denied.
- Things to consider / constraints:
    - If the file does not exist, then an appropriate error message should be displayed and logged.
    - A user must be logged in to execute this command

`ls` *`filename`*

- Example: `ls file.txt`
- Displays and logs the owner, group, and permissions for a file. An example output of this instruction would be
  `file.txt: tom students rwx rw- r--`
- Things to consider / constraints:
  - If the file does not exist, then an appropriate error message should be displayed and logged.
  - Any user may execute this command on any file regardless of permissions.

`end`

- Example: `end`
- The last instruction in the input file.
- Indicates that your program should write out the *groups.txt* and *files.txt* files as described previously, perform any other necessary cleanup actions (i.e., closing open file streams) and terminate.

Extra credit (up to 15 points)

In the *accounts.txt* file store users' passwords as hash functions (using `openssl`) instead of as plain text. When a user is created, run a hash algorithm on its password (think Homework 1), and save the hash (instead of the plain text password) in the *accounts.txt* file. When a user tries to log in, hash the given password and compare it to the one stored in the *accounts.txt* file. If you chose to implement this functionality, your program should NOT require any modifications to the syntax for ANY of the instructions.

To get bonus credit, your program MUST utilize the external Linux command `openssl` with appropriate arguments. It should NOT use hashing functionality that is built into the programming language that you choose.

HINTS:
Look up how to run shell commands from inside a program. Also, there is an `openssl` argument that you can use that will tell `openssl` to output the hash in a hexadecimal string format. This is likely the option that you will want to use.