CS5950 Programming Assignment #1: User-level Access Control Lists

Due Date: *Friday, October 2, 2015 @ 11:59pm*

The UNIX file system protection scheme does not allow for fine-grained control over access using only the standard user(u), group(g), and others(o) protection bits. For example, its not possible using only these bits to grant one set of users read access, a second set of users write access, and a third set of users both read and write access. Further, when the group bits may suffice, using these access bits typically requires intervention by the super-user. However, the SETUID and SETGID bits can sometimes be used to provide finer-grained control when it is desired. In this assignment, you will create `get` and `put` commands that allow a user to provide a finer-grained control over access to his files without the intervention of the super-user.

Overview

The basic idea is that a file owner will dictate access to his file named `basename.ext` by specifying users that are allowed access and the type of access each user is allowed in a file named `basename.ext.access`. Here `basename.ext` represents an arbitrary file name and `basename.ext.access` is the access control list for file `basename.ext`. Users gain read access to the files via the SUID binary `get` (that you will write) and which the file owner will place in an appropriate location. Write access is gained via the `put` binary, which you will also write. If the file `basename.ext.access` does not exist, no access is allowed via `get` or `put`.

Requirements

Access Control. Access to the protected file `basename.ext` is determined by the contents of the ACL file named `basename.ext.access`. If the ACL file does not exist, both `get` and `put` exit silently. Entries in the ACL file each contain two components separated by whitespace (space, tab). The first component, which may be preceded by whitespace, is a single userid (alphanumeric value, e.g. "carr"). The second is a single character `r`, `w`, or `b`, indicating read, write, or both read and write access, respectively, for the user with the corresponding userid. This second component may be followed by whitespace. Lines beginning with the character '#' are comments. No blank lines are allowed. `Get` and `put` check for malformed entries before beginning operation and existence of a malformed entry causes a silent exit. **A silent exit must always emit the phrase "silent exit" to the terminal before exiting.** If the ACL file is a symbolic link, `get` and `put` exit silently. If the protection for `basename.ext.access` allows any world or group access (via the standard UNIX file protections), `get` and `put` fail silently. If the protected file `basename.ext` is not an ordinary file, `get` and `put` fail silently.

Access. A file owner allows access to his files by placing a copy of `get` and `put` in an appropriate directory, setting the SUID bit, and allowing others to execute the binary. From the perspective of `get` or `put`, the files whose ownership is specified by the effective uid of the executing process are being protected. The files are being protected against the user whose uid corresponds to the real uid of the executing process. For the discussion below, `owner` is the owner of the binary (*i.e.*, `get` and `put`) and `user` is the user of the binary and the one getting access to `owner`'s files

A `user` attempts read access to a file by executing the command

```
get <source> <destination>
```

`Get` determines the ownership for `source` and `destination` before performing the operation. (See the manual page for `fstat().`) Access is allowed only if

- `source` is owned by `owner`,
- `owner` has read access to `source`,
- the file `source.access` exists and contains an entry granting `user` read access
- `source.access` must be owned by `owner` and have owner permission only – no group or world access
- `user` can write to the file `destination` and owns that file if it exists, and is made the owner if it does not exist.

If read access is allowed, the file `source` is copied to the file `destination`. If `destination` already exists, the user is queried before the file is overwritten.

A user attempts to write a file by executing the command

```
put <source> <destination>
```

`Put` determines the ownership for `source` and `destination` before performing the operation. (See the manual page for `fstat().`) Access is allowed only if

- `owner` owns `destination`,
- `owner` has write access to the file `destination`,
- the file `destination.access` exists and contains an entry granting `user` write access
- the file `destination.access` must be owned by `owner` and have owner permission only – no world or group access, and
- `user` may read `source`.

If write access is allowed, the file `source` is written to the file named `destination`. If `destination` already exists, the user is queried before the file is overwritten. If `destination` is overwritten, the owner and protections of the file are not changed by the write. If `destination` does not exist, it is created with the owner and group corresponding `owner` and his default group. (See the manual page for `getpwnam().`) The file protection is set to 400.

Miscellaneous. You need not worry about file locking for this assignment. You may assume that only one instance of `get` or `put` is operating against a file at any given time. **The rules discussed for secure SUID programming must be followed in this assignment.** The project must be coded in C and will be tested on a Linux system similar to the one you will be given in a virtual machine.

Collaboration Rules

This project may be performed in pairs or by a single person. Of course, there are no restrictions on interactions among group members. However, each group must work independently. A group may neither show any other its code nor look at the code of another group. (This policy extends to any external resource, including code found on the web or individuals who are not enrolled in the course.) Individual work will be allowed, but no allowance will be made in the due date or other submission requirements if this option is freely chosen.

Kali Linux

The virtual machine for Kali Linux can be found at `https://jjohnson.cs.wmich.edu`. The download link is labeled VM Download under CS5950 VM Download header. This is a `.ova` file for VirtualBox so you will use VirtualBox Appliance Import Wizard to import the VM. The link below contains step by step directions on how to do this after you have downloaded the `.ova` file `http://docs.oracle.com/cd/E26217_01/E26796/html/qs-import-vm.html`. The user for Kali is `root` and the password is `Changem3`. You should change the password for `root` and also create users for your self and your partner. If you are doing this project by yourself. You will want to create a second user for testing. In either case, you can create a group for both users that allows you to share files. If you do not know how to create users and groups, please see the CS systems administrators in room C-218.

Submissions

You must prepare a `makefile` and all necessary source files so that I can simply do a `make` and build `get` and `put`. To that end, create a directory called `project1` in which your `makefile` and all required source files will reside. Make `project1` your working directory and tar the contents of the directory with "`tar zcvf project1.tgz *`". Submit the assignment via Canvas. Each pair (or individual) must send me their names by Monday, January 26, 2014 at 5:00pm. With your submission, include a README file that gives the names of each person who worked on the submission and an outline of each person's contribution to the completed program. Your code should be well commented. In addition to normal documentation, include comments in your code at points related to the security requirements (e.g. `/* Changed effective uid back to real .....  */`). Also, in the README file, give an overview of your implementation and identify and defend any security-related decisions you had to make during the implementation.