CS483 Project #1 - Access Control

Due Date: Friday, October 13, 2017 @ 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 their 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

The contents of the ACL file named `basename.ext.access` is used to determine whether access to protected file `basename.ext` is allowed. 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. "pjbonamy"). 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. 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 their 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.

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 the effective uid of the executing process,
- the effective uid of the executing process has read access to `source`,
- the file `source.access` exists and indicates read access for the real uid of the executing process,
- and the real uid of the executing process can write the file `destination`.

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. The real uid of the executing process should be the owner of `destination`. The file protections may assume the default values.

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

- the effective uid of the executing process owns `destination`,
- the effective uid of the executing process has write access to the file `destination`,
- the file `destination.access` exists and indicates write access for the real uid of the executing process, and
- the real uid of the executing process 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 to the effective user of the executing process and their 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 should be followed in this assignment. The project should be coded in C and will be tested on a Linux system.

Submissions

You must prepare a `Makefile` and all necessary source files so that I can simply do a `make` and build `get` and `put`. Your code must be neatly formatted, and include comments about general structure and points related to security requirements (e.g./* Changed effective uid back to real ..... */). Also include a README file which provides an overview of your implementation and identifies and defends any security-related decisions you had to make during the implementation. Package all of your files into a single zip file or tarball, and submit the file via Blackboard.