# Assignment 3

Due Sunday October 6, 2019 11:59pm via sakai

See the FAQ

## Introduction

Access control mechanisms in operating systems have evolved over the years to include access control lists as well as a variety of mandatory access control (MAC) mechanisms, including multi-level security, integrity levels, type enforcement, and limited forms of role-based access control.

An operating system, however, can only deal with the users and resources it knows about. It manages access between subjects (users) and objects (resources provided by the system, such as files and devices). There's an underlying assumption that these subjects (users) have accounts on the system and the objects are known to the system.

For many applications, however, this is not the case. Applications may run as services that are launched by a specific user. These services, in turn, interact with users who may very well not have accounts on the system. For instance, you can log onto eBay and interact with it but you don't have an account on any of the systems that provide the eBay's service. Similarly, objects may be entities that are also unknown to the operating system, such as fields or tables in a database or media streams.

This is a problem that affects many environments. Services often have to put together their own solutions to manage their user accounts and access permissions (authorizations). To address this, Google recently built Zanzibar: Google's Consistent, Global Authorization System. This provides a consistent, large-scale service for managing access control policies that any application can use. Google uses this for services that include Calendar, Cloud, Drive, Maps, Photos, and YouTube.

## Your Assignment

Your assignment is to design and implement an authentication and access control (authorization) library that can be used by services that need to rely on their own set of users rather than those who have accounts on the computer.

The access control system that you design will support:

*Users*

A collection of users and passwords that is used for authenticating users and for members of user groups. For example

{ "anika", "password" }, { "fang", "123456"}, { "liam", "abc123" }, …

*User groups*

A named collection of one or more users. For example,

```
admins = { "anika", "arun", "wei", "yash" }

premium_subscribers = { "fang", "noah", "riya" }

normal_subscribers = { "liam", "ravi", "olivia" }
```

*Object groups*

A named collection of one or more objects. Objects are any strings that will have meaning to the application. For example, they might be file names, directory names, subscribed features, media streams, etc. For example,

```
premium_content = { "hbo", "showtime", "disney" }

normal_content = { "cbs", "nbc", "fox", "abc", "wor", "pix", "pbs" }
```

*Access permissions*

A set of access rights that defines operations that **user groups** can perform on **object groups**. For example,

```
"view": (premium_user, premium_content)

"delete": ("admins")
```

Note that the *delete* operation does not specify any objects but rather just an operation that those in the user group *admins* may perform.

The above examples are conceptual. It is up to you to decide what storage structures are the most convenient to implement.

## Groups

You may work on this assignment individually or in a group of up to three members. If you work in a group, please submit only one version. If you are not doing this as an individual project, the standards for grading will be more stringent.

## Languages

You may write this assignment in C, C++, Go, Java, or Python.

## Environment

Your submissions will be tested on Rutgers iLab Linux systems. You can develop this on any other system but you are responsible for making sure that it will work on the iLab systems.

## Specifications

Your implementation will be one that runs locally rather than as a network service and can be incorporated within any application. For this assignment, you do not need to handle any concurrent operations, so you need not implement locking.

Applications that use this service can be assumed to be trusted and trustworthy: they will not try to use the interfaces incorrectly or subvert the system in any way.

User accounts and access permissions will be stored persistently in one or more files so that they can be accessed again when the application (or another program using the same authorization service) is run again.

You will write and submit several programs, each of which demonstrates a specific function of the interface. These programs also make your program suitable for use and testing by shell scripts.

## API

This section describes the operations that you need to implement and the program that uses them. It is up to you to define the appropriate return values, exceptions, and other details that you feel are needed for your design.

### *AddUser("user", "password")*

Define a new user for the system along with the user's password, both strings.

**Test program**:

```
AddUser myname mypassword
```

The program should report an error if the user already exists.

### *Authenticate("user", "password")*

Validate a user's password by passing the username and password, both strings.

**Test program**:

```
Authenticate myname mypassword
```

The program should clearly report

- Success
- Failure: no such user
- Failure: bad password

### *AddUserToGroup("user", "groupname")*

Add a user to a user group. If the group name does not exist, it is created. If a user does not exist, the function should return an error.

**Test program**:

```
AddUserToGroup user usergroupname
```

The program should report

- Success & list all the users in that group
- Failure if the user does not exist

### *AddObjectToGroup("objectname", "groupname")*

Add an object to an object group. If the group name does not exist, it is created. The object can be any string.

**Test program**:

```
AddObjectToGroup object objectgroupname
```

The program should report

- Success & list all the objects in that group

### *AddAccess(“operation”, “usergroupname”, “objectgroupname”)*

Define an access right: a string that defines an access permission of a user group to an object group. The access permission can be an arbitrary string that makes sense to the service.

**Test program**:

```
AddAccess operation usergroupname [objectgroupname]
```

The program will accept two or three strings. If objectgroupname is missing, it is considered null and the specified user group is simply permitted access to the operation regardless of the object (or an object may not make sense for that operation).

### *CanAccess(“operation”, “user”, “object”)*

Test whether a user can perform a specified operation on an object. Optionally, an object may be NULL, in which case *CanAccess* checks allows access if a user is part of a group for an operation on which no object group was defined.

**Test program**:

```
CanAccess operation user [object]
```

The program will check whether the user is allowed to perform the specified *operation* on the object. That means that there exists a valid *access* right for an *operation* where the user is in *usergroupname* and the object is in the corresponding *objectgroupname*.

As with *AddAccess*, the program will accept two or three strings. If object is missing, it is considered null and the software allows access only if no object groups were defined for that *{operation, usergroupname}* set.

Note that the parameters here are user names and object names, *not* user groups and object groups.

### Notes and assumptions

Note that this is not a complete interface. It is, for example, notably missing operations to delete or change users, user groups, object groups, and permissions.

For simplicity of implementation, you may assume that a user is in exactly one group and an object is exactly in one group. To handle the case of an empty object group, you may store a placeholder string, such as “`null`” if that simplifies your implementation. You may do the same with the object name.

**Extra credit:** Design your program so that this assumption need not apply: a user may be a member of multiple groups and you need to check all appropriate access rights to see if a user is allowed access to an object.

### Hard-coding paths

Because the program will be run from other accounts, it is imperative that you **do not** include hard-coded full pathnames in your program (e.g., do not use a rooted pathname like “`/ilab/users/pxk/src/access/tables`”). You may store all your files in the current directory or a subdirectory (e.g., “`./tables/`”) so that cleanup will be easy.

## What to submit

Your submission will generate six programs: **AddUser**, **Authenticate**, **AddUserToGroup**, **AddObjectToGroup**, **AddAccess**, and **CanAccess**.

### Documentation

Documentation is crucial so that we don't waste time trying to figure out how to compile and run your program. At a minimum, specify clearly:

1. How to compile your programs (a **Makefile** would be useful). There should be **NO** reliance on any IDE (e.g., eclipse) or third-party libraries. Instructions should specify command-line commands only.
2. Any setup that is needed, such as creating a subdirectory to hold your access permission files.
3. The usage of each program, including examples.
4. Example test scripts that you used to validate the program, testing error cases as well as success cases.

If you are submitting a group project, also write up a discussion of your implementation, including how you store the data for implementing this system.

## Code

Submit only your documentation and source files (and Makefile, if you have one). **DO NOT** submit any compiled files (e.g., object files, Java `.class` files or executables).

Make sure the instructions you submit to compile and run the programs makes sense. Try to follow them based on a clean download of what you submitted. Better yet, have a friend try to follow them.