# Project 1

Start Assignment

---

**Due** Feb 20 by 11:59pm      **Points** 10      **Submitting** a file upload

**Available** Feb 3 at 12am - Feb 20 at 11:59pm 18 days

---

See the **FAQ**   **(https://people.cs.rutgers.edu/~pxk/419/hw/p-1-faq.html)**

# 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 and managed by the system (e.g., files in the file 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 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 services. Services often have to put together their own solutions to manage their user accounts and access permissions (authorizations). To address this problem, Google built **Zanzibar: Google's Consistent, Global Authorization System (https://www.usenix.org/system/files/atc19-pang.pdf)**. 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 and manage their own sets of objects and operations on those objects.

The access control system that you design will support:

**Users**

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

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

## Domains

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

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

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

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

## Types

A named collection of zero 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 **domains** can perform on **types**. For example,

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

```
"delete": ("admins", normal_content)
```

```
"delete": ("admins", premium_content)
```

The access rights are meaningful only for the application that is making use of this service. That is, *view* and *delete* in these examples are arbitrary strings. Note that the *delete* operation does not specify any objects but rather just an operation that those in the user group *domains* may perform.

▶

This system is similar to SELinux's (Security Enhanced Linux) type enforcement system.

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

# Groups

This is an individual assignment. You may not collaborate in your development of this project.

# 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. You should check for basic usage (correct number of arguments or empty strings) but you can assume that anyone using your code for managing access control will not

You will write a single driver program named **auth**.

This program will take commands as parameters, each of which implements a specific function of the interface. The program should be suitable for use and testing by shell scripts. This means should follow the defined interface and not generate extraneous output.

The program gets all the parameters it needs from the command line and exits after running the requested command. **It will never prompt for user input or read data from the standard input.** User accounts and access permissions **must 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. If you choose to use a structure like a Python dictionary, you are responsible for storing it into a file before the program exits and reading it in when the program runs.

▶ number of arguments is incorrect or if the command parameter is invalid, you should print a descriptive error message and exit. Error messages start with the string `Error:` followed by a space and a one-line error message.

# API

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

Each of these functions and their parameters will be invoked from the main program and each of the parameters will be a single argument on the command line. For example,

```
auth AddUser paul "monkey brains"
```

is the command to add a new user named `paul` with the password `monkey brains`.

Responses are either:

- A single line with the text `Success` to indicate a successful operation if there is no other output (i.e., a list of items) that needs to be presented.
- A single line starting with `Error:` followed by an optional message separated from the `Error:` string with a space to indicate what went wrong.
- A list of items, one item per line.

# AddUser("user", "password")

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

The username cannot be an empty string. The password may be an empty string.

**Test program**:

```
auth AddUser myname mypassword
```

The program should return one of:

- `Success`
- `Error: user exists` - if the user already exists.
- `Error: username missing` - if the username is an empty string.

# Authenticate("user", "password")

▶

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

**Test program**:

```
auth Authenticate myname mypassword
```

The program should clearly report

- `Success`
- `Error: no such user`
- `Error: bad password`

# SetDomain("user", "domain")

Assign a user to a domain. Think of it as adding a user to a group.

If the domain name does not exist, it is created. If a user does not exist, the function should return an error. If the user already exists in the domain, the command will succeed but take no action. You should never have duplicate users in a domain.

A user may belong to multiple domains.

The domain name must be a non-empty string.

**Test program**:

```
auth SetDomain user domain_name
```

The program should report

- `Success`
- `Error: no such user` - if the user does not exist
- `Error: missing domain` - if the domain name is an empty string. Your program should check that the domain is a valid name before checking that the user exists. That avoids having to check for a user only to find out that the domain name is an empty string.

# DomainInfo("domain")

List all the users in a domain.

The group name must be a non-empty string.

**Test program**:

```
auth DomainInfo domain_name
```
▶

The program should output:

- A list of all the users in that domain, one per line
- Nothing if the domain does not exist or there are no users in a domain (empty list). A non-existent domain name is treated the same as one with no users in it.
- `Error: missing domain` - if the domain name is an empty string.

Sample output:

```
alice
bob
charles
david
ellen
```

with no leading tabs or spaces. This output format fits into the Unix tools philosophy, which makes the output suitable for input in a pipeline of commands.

# SetType("objectname", "type")

Assign a type to an object. You can think of this as adding an object to a group of objects of the same type.

If the type name does not exist, it is created.

The object can be any non-null string.

**Test program**:

```
auth SetType object type_name
```

The program should report

- `Success`
- `Error:` Failure if the object or the type names are empty strings.

# TypeInfo("type")

List all the objects that have a specific type, one per line.

The type name must be a non-empty string.

**Test program**:

```
auth TypeInfo type_name
```

▶ program should output

- A list of all the objects that have been assigned `type_name` with `SetType`.
- Nothing if the type does not exist or there are no objects associated with that type (empty list).
- `Error:` if the type name is an empty string.

Sample output:

```
alice_file
bob_file
charles_file
david_file
ellen_file
```

with no leading tabs or spaces. This output format fits into the Unix tools philosophy, which makes the output suitable for input in a pipeline of commands.

# AddAccess("operation", "domain_name", "type_name")

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

The domain name and type name must be non-empty strings.

If the domain name or type names do not exist then they will be created. If the operation already exists for that domain and type, it will not be added a second time but it will not be treated as an error.

This is the key function that builds the access control matrix of domains and types.

**Test program**:

```
auth AddAccess operation domain_name type_name
```

The program will accept three strings and report

- `Success` - if the operation was added to the access control matrix
- `Error: missing operation` - if the operation is null
- `Error: missing domain` - if the domain is null
- `Error: missing type` - if the type is null

# CanAccess("operation", "user", "object")

Test whether a user can perform a specified operation on an object.

The logic in this function in pseudocode is:

```
for d in domains(user)
 ▶       for t in types(object)
                if access[d][t] contains operation
                        return true
return false
```

That is, find all domains for the *user* and find all types for the *object*. Then see if any of those domain-type combinations have the desired *operation* set.

**Test program**:

```
auth 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* in some (*domain*, *type*) where the user is in *domain* and the object is in the corresponding *type*.

As with *AddAccess*, the program will accept three strings.

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

The program will return:

- `Success` - if the access is permitted

- `Error: access denied` - for all other cases. This means the operation does not exist for that any domains for that user-object or the user does not exist. `CanAccess` provides the answer to the question, *can user A access object B?* If the operation does not exist, the answer is no. You don't need to provide an explanation of the error.

# Command syntax error handling

Your program should print an error if an invalid command is entered or an incorrect number of arguments is specified. For example:

```
auth Add myname mypassword
Error: invalid command Add

auth Authenticate myname mypassword mypassword2
Error: too many arguments for Authenticate
```

# Notes and assumptions

Note that this is not a complete interface. It is, for example, notably missing are operations to delete or change users, user groups, object groups, and permissions. It also has no support for wildcards or regular expression patterns.

▶ Design your program so that a user may be a member of multiple domains and an object may have multiple types associated with it. This means that you will need to check all appropriate access rights to see if a user is allowed access to an object.

Because this is implemented as a program that accepts a single command at a time, you will need to save your results in one or more files after running each command. **Saving and reading your lists and tables may be the most time-consuming part of designing your program**. Think about good ways to implement this so your code can remain simple.

# Hard-coded 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 your file in the current directory or, if using multiple files, in a subdirectory (e.g., "`./tables/`") so that cleanup will be easy. Create the directory if it's missing. Don't assume that anything is set up before we run the program.

# What to submit

Your submission will generate one program, named auth, that takes the following commands: **AddUser**, **Authenticate**, **SetDomain**, **DomainInfo**, **SetType**, **TypeInfo**, **AddAccess**, and **CanAccess**.

# Documentation and components

Clear and simple 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 build script or **Makefile** would be useful). There should be **NO**reliance on any IDE (e.g., eclipse) or third-party libraries (with the exception listed in the next section). 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. Example test scripts that you used to validate the program, testing error cases as well as success cases.

# 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).

▶ feel a need to do so, you may include third-party support **only** for serializing or parsing stored
If the packages are not installed on Rutgers systems, you will need to submit the source components and make sure they are integrated into your build script.

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.

Note that there are 150 students in this class. With a smaller class, we would be able to devote a several minutes to figuring out how to compile and run your program. With a class this size, you will need to make sure it is trivial to build your program and that it conforms to the interface.