W0: I will be utilizing an RBAC based language implementation of a simple access control system. This implementation supports indirection and assignment of any size of csv file in a format specified in W1. Delegation is supported in the form of roles, or groups; users can be delegated into groups defined by separate entities to the users, and the group permission attributes are inherited. Another feature of this implementation is nested group inheritance. See W5 for a specific example, but in general, this means that a user’s group can be part of a group, and the higher level group implicitly inherits the lower level groups permissions, but not the other way around, for the sake of security. The second additional feature is role intersection in the context of RBAC. This implies that a user’s role can be utilized to indirectly reference the user and the permissions inherited from the role, and there is a subsequent intersection between the permissions that role has hierarchically inherited from a role above itself are.

W1: This implementation of Role Based Access Control policy is done within the scope of a python shell, and its interactions with the main .csv file containing the list of all groups and permissions. The structure of this file is as follows: either, “type, name, data, permission”, or “type, name, role”. “type” can be either g or p, implying policy or group. The parser within the python script reads in the .csv file and organizes the aforementioned types of data lines into two batches. The batches are represented by two 3-dimensional arrays, with the structure being each index contains a list, which represents an indexed version of a line from the .csv file. Next, when prompting the user for name, data, and read/write, the script first searches the policy basket, then it searches the group basket, and lastly it searches for groups of groups related to the user to determine if the user has permission to read or write the file. This is accomplished with a main while loop for prompting user input, then nested if, elif, else statements that search through the two 3-dimensional array baskets to find the prompted permission. This is accomplished by first searching for user level permissions in the permission basket, and if that is unsuccessful, checking the groups for the user’s name and then checking a matched group’s permissions, and if that fails, checking for a higher level (hierarchical) group above that matched group for the inherited permission. If all of these fails, then we deduce that the individual user does not have permission to either read or write the given file/data. The final two levels of the search include the implementation of role hierarchy and role intersection. The hierarchical inheritance is checked as a last resource, as it would be the most computationally intense relative to standard user permission in the form, “ p, user, data, read/write”. Also, role intersection is seen in the delineation between the group and permission basket, and the interaction in searching after user prompted permissions. Examples of both can be seen in W5.

C2: See *C2.py* and *policy.csv*

W3: *Indirection* can be seen if we view groups(roles) as dereferenced objects that have attributes such as permission, as seen in the following examples…

In the examples below, you will see that you can add a line “g, user, file” to the csv, which gives the user indirect access to certain files, as they are an administrator of that file, even without being directly written into the policy.

example1.csv

…

p, alice, data1, read

p, bob, data2, write

p, data2\_admin, data2, read

p, data2\_admin, data2, write

g, alice, data2\_admin

…

example1.py

…

e = casbin.Enforcer("model.conf", "policy.csv")

if e.enforce(“alice”, “data2”, “write”):

print(“alice must be a data2 admin!”)

…

example2.csv

…

p, alice, data1, read

p, bob, data2, write

p, data3\_admin, data3, read

p, data3\_admin, data3, write

g, bob, data3\_admin

…

example2.py

…

e = casbin.Enforcer("model.conf", "policy.csv")

if e.enforce(“bob”, “data3”, “read”):

print(“bob must be a data3 admin!”)

…

W4: *Delegation* can be seen in the form of *Roles*, as seen in the following examples…

Delegation involves roles, or administrative groups, similar to the above examples. You can delegate a team of users to have, for example, have access to read a file “secret”, when noone else in the policy system has access. Here are two examples of adding delegated roles.

example1.csv

…

p, secret\_admin, secret, read

g, tim, secret\_admin

g, allen, secret\_admin

g, zoey, secret\_admin

g, stan, secret\_admin

…

example2.csv

…

p, userbase\_admin, users, write

g, bob, userbase\_admin

g, james, userbase\_admin

g, stacy, userbase\_admin

g, carol, userbase\_admin

…

W5: Role Hierarchy and intersection are two additional access control features included in the access control protocol.

There can be a level of nested roles as seen in example 1. Jan is a data\_admin, and data\_admin shares all of data2\_admin’s permissions because it is a level above in hierarchy, so even though we don’t list explicitly that data\_admin can read/write data2, it is encoded in the hierarchy, therefore jan has access to read/write data2.

RoleHeiarchyExample1.csv

…

p, alice, data1, read

p, bob, data2, write

p, data2\_admin, data2, read

p, data2\_admin, data2, write

p, data\_admin, data1, read

p, data\_admin, data1, write

p, data\_admin, data3, read

p, data\_admin, data3, write

g, alice, data2\_admin

g, jan, data\_admin

g, data\_admin, data2\_admin

…

OutputExample1.txt

…

>>User: jan

>>File: data2

>>read/write: read

jan has read privilege for data2 via data\_admin -> data2\_admin

…

RoleHeiarchyExample2.csv

…

p, alice, users1, read

p, bob, users2, write

p, user\_db\_3, users3, read

p, user\_db\_3 users3, write

g, alice, user\_db\_3

g, stan, user\_db\_admin

g, user\_db\_admin, user\_db\_3

…

OutputExample1.txt

…

>>User: stan

>>File: users3

>>read/write: read

stan has read privilege for users3 via user\_db\_admin -> user\_db\_3

…

intersection can be seen within the scope of role assignment, as a user’s groups permissions, as well as explicit user permissions are intersected when the query is processed. We see examples of this when you compare the output of referencing a individual user whose specific permissions are encapsulated in their role, to the output of referencing the role permissions itself.

intersectionExample1.csv

…

p, alice, users1, read

p, bob, users2, write

p, user\_db\_3, users3, read

p, user\_db\_3, users3, write

g, alice, user\_db\_3

g, bob, user\_db\_admin

g, user\_db\_admin, user\_db\_3

…

intersectionOutput1.txt

…

>>User: alice

>>File: users3

>>read/write: read

alice has read privilege for users3 via user\_db\_3

…

>>User: user\_db\_3

>>File: users3

>>read/write: read

user\_db\_3 has read privilege for users3

…

intersectionExample2.csv

…

p, alice, users1, read

p, bob, users2, write

p, user\_db\_3, users3, read

p, user\_db\_3, users3, write

g, alice, user\_db\_3

g, bob, user\_db\_admin

g, user\_db\_admin, user\_db\_3

…

intersectionOutput2.txt

…

>>User: bob

>>File: users3

>>read/write: write

bob has write privilege for users3 via user\_db\_admin -> user\_db\_3

…

>>User: user\_db\_admin

>>File: users3

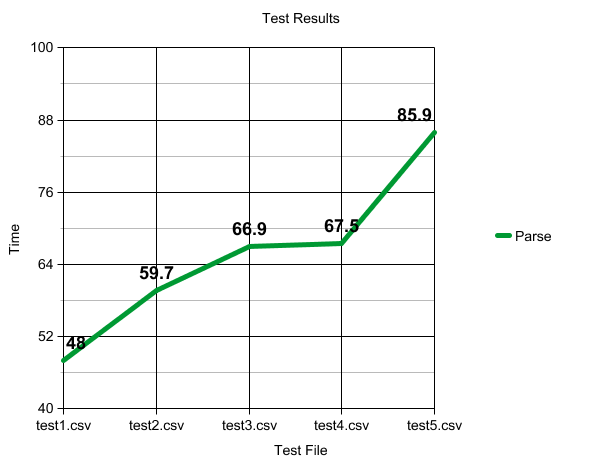
>>read/write: write

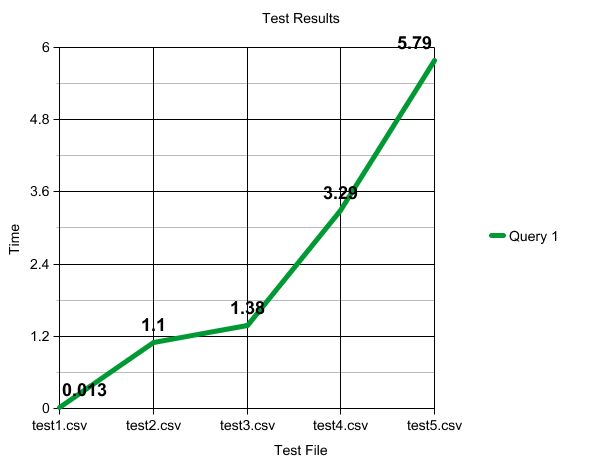
user\_db\_admin has write privilege for users3 via user\_db\_3

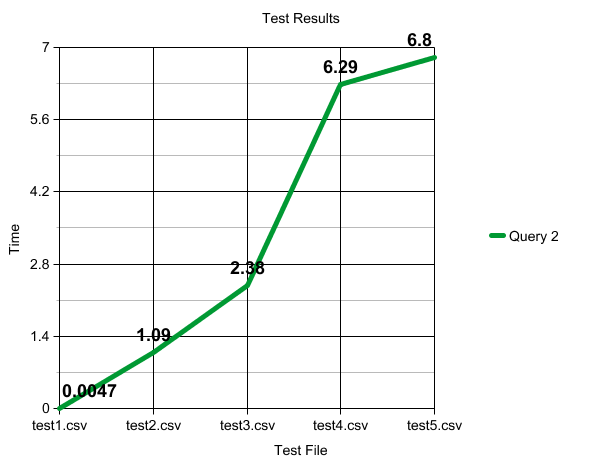
…

C6: See *C6.py, test1.csv, test2.csv, test3.csv, test4.csv, test5.csv*

W7:







The above tests reflect the parsing and processing, then two queries of each of the individual test files. The policies increase in both size and complexity starting from test1 to test5. C6.py automatically computes these tests, and times separately the parsing and the computation. The results have been represented, for each of the test files, the the above graphs. The processing is obviously going to increase as the files get larger, but it is not necessarily a linear increase, as the files are not following a specific ratio of growth, and also we must consider that when more complex roles are added, this slightly increases the time it takes to process and parse the policy. With that being said, the general increase in time as we move form test1 -> test5 is close enough to what would be predicted, however it would be more linear if there were an exact linear change in file size for each test. Next, when we look at the two queries performed for each test file, the time is relatively low for test1 to test3, and afterwards we see a far steeper increase in time taken for the queries. This is because of multiple factors, most notably the increase in group complexity, and size of files after test3. The former can be explained simply by fact that nested groups require more time to track and find permission for, and the ladder can be rationalized by considering the fact that a larger policy, regardless of parsing time, still needs to be traversed logically when looking for user permissions, specifically if they are role inherited permissions. These results show us that this language specifically would not likely scale well with immense, complex policies, as it would simply take too much time to work at the scale needed to run something such as a large cloud storage service. I think the REBAC implementation is a good one, but much more complexity and nuances would need to be added to handle such an industrial, enterprise grade access control scheme. This language, however, would likely work very well for personal, local use, as the time constraints would not be noticeable. This project has taught me that adding layers of complexity and features are difficult in the scope of access control enforcement, and it would likely be difficult to balance the requirements of many features, and efficiency when building an access control enforcement scheme for an enterprise level system.