

Note: instructions for setup and running application are at the end of the report

Task W0

I chose to implement my access control system within a Flask web application. I felt that this would be an interesting choice because it resembles a real-world use. My particular application is used to process information about members of the University of Pittsburgh. For my access policy, it seemed logical to me for administrators to enter user information and then have the system automatically store that information appropriately. Based on the attributes defined by the admins, the access control system creates databases which are used to answer queries about users. My application supports queries for indirection, delegation, attribute intersection, and attribute inference.

Task W1

On the backend I used Python, Flask, SQL Alchemy, and JSON to implement my access control policy. Before running the application, the database and user information is processed using the “flask initdb” command. It begins by loading the information for university members from the JSON file (access_policy.json).

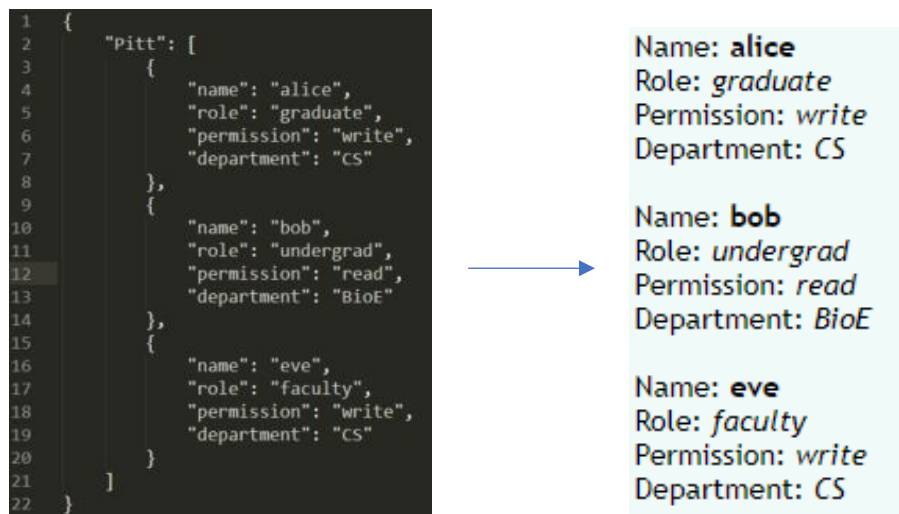


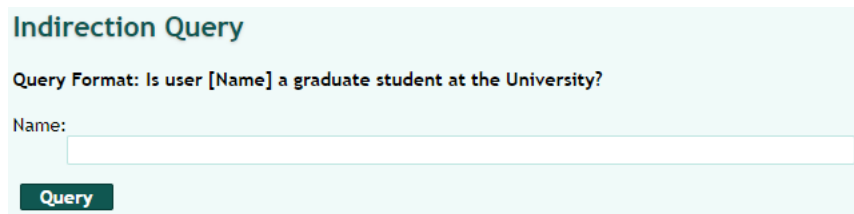
Figure 1: Example JSON file where user attributes are defined (left) and how it is displayed to user after being parsed by the access control system (right)

The users and their attributes (shown in Figure 1) are stored in lists and dictionaries. Each user is stored in a dictionary and inside that dictionary, each attribute (left of colon) is set to the value specific to that user (right of colon). The app then parses the JSON file and creates databases to store the information in the create_access_policy() function implemented in access.py. For this particular example, I chose to make a Pitt database contain all university members, a Graduate database to group all graduate students, and a CS database to group all CS members and to act as a distinct entity. To support the required features, I made 4 separate functions (indirection(), delegation(), intersection(), and inference()) that implement the access control features presented to the user by the client. To implement access control, each function performs a database query specific to that access feature and then informs the user about the result.

Task W3

Indirection is useful when permissions are assigned in groups. For my specific case, we can assume that graduate students will have a specific set of permissions. For example, all graduate students have write access on the university file system, and all graduates have free access to MATLAB. To represent this relationship, I created a Graduate database that links each graduate student with their attributes stored in the overall Pitt database. In practice, the Graduate

database could assign any variety of additional attributes. To demonstrate this feature, I used a query that finds the `pitt_id` of the name entered by the user, then checks it against the Graduate database to see if the person entered is a member of the graduate group and would therefore have graduate privileges. The client-facing Indirection Query utility is shown in Figure 2.

The image shows a web interface titled "Indirection Query". Below the title, it says "Query Format: Is user [Name] a graduate student at the University?". There is a text input field labeled "Name:" and a green button labeled "Query".

Indirection Query

Query Format: Is user [Name] a graduate student at the University?

Name:

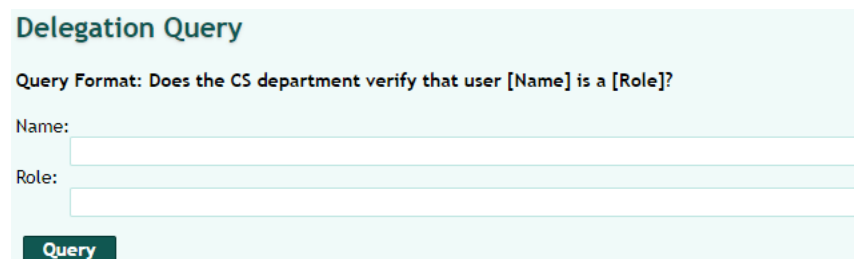
Query

Figure 2: Indirection query utility used to check Graduate group

For example, querying the indirection utility for bob will report that he is not a graduate student, while querying the indirection utility for alice will report that she is a graduate student.

Task W4

Delegation is an access control mechanism by which we consult a third party to decide on user privileges. In my case, let's assume the university wanted to know whether a particular person was faculty in the CS department. Rather than looking through its own records, the university could consult the CS department instead. To determine this access condition, my application queries the CS department to see if the name of the person entered by the user is in fact a member. Then my application queries against that user's attributes to see if their role is faculty. If we used the Delegation Query (shown in Figure 3) utility to check whether alice was a faculty in CS, the application would inform us that access is denied because the user does not meet the criteria. However, if we perform the same query with eve instead, access is granted.

The image shows a web interface titled "Delegation Query". Below the title, it says "Query Format: Does the CS department verify that user [Name] is a [Role]?". There are two text input fields, one labeled "Name:" and one labeled "Role:", and a green button labeled "Query".

Delegation Query

Query Format: Does the CS department verify that user [Name] is a [Role]?

Name:

Role:

Query

Figure 3: Delegation Query for consulting the CS department

Task W5

One of the optional features I chose to implement is attribute intersection. This access relationship arises when members of two separate groups can fulfil the requirement. In my scenario, the group consisting of graduate students and the group consisting of CS student both have free access to MATLAB. When the user enters the name of the person they want to check, my access control system queries the CS department to see if that person is a member of their department. If they are, the application informs the user. If they are not, the system then queries the group of graduate students to check whether the person is a member of that group. Again, if they are, the application informs the user. If they are not, the application informs the user that the person does not have free access to MATLAB. Using the Attribute Intersection Query utility (shown in Figure 4) we would see that eve is a member of the CS department and therefore has access. We would also see that even though john is not a member of CS, he is a graduate student in another department and therefore has access. Finally, we would find that bob is neither in CS nor is he a graduate student and he therefore does not have access to MATLAB.

Attribute Intersection Query

Query Format: Does user [Name] have free access to Matlab?

Name:

Query

Figure 4: Attribute Intersection Query for checking distinct user groups

The second optional feature I chose was attribute inference. This access control mechanism is using one attribute to infer something about a different attribute. For my scenario we assume that undergrads have read only access on the university file system. My access control system will query the permissions granted to the person that the user chooses. Then if we find that the user has read only permissions, we can infer that they are an undergrad. Using the Attribute Inference Query utility (shown in Figure 5) we would find that we cannot infer that alice and eve are undergrads because they have write permissions meaning they could be either graduate students or faculty. However, bob does have read permissions which means he must be an undergrad.

Attribute Inference Query

Query Format: Is user [Name] an undergrad?

Name:

Query

Figure 5: Attribute Inference Query for inferring academic standing based on permissions

Task W7

Studying the efficiency of my access control policy with varying sizes of input was relatively straight forward. I created a second JSON file with 60 people instead of the 4 people I used for the demo. I had to manually create each entry so scaling to huge numbers of users in the JSON file was not realistic. To switch between the two, I simply changed the filename being parsed. I measured the time before parsing the JSON file and creating the database, then I measured the time after and found the difference. For the small user size of 4, I found that the policy creation only took about 1 millisecond on average. However, for 60 users the average control policy system creation time was 12 milliseconds on average. This may not seem like a big deal because 12 milliseconds is still not long, but this has worrying implications for scaling my access policy control system. My system would struggle to scale efficiently because the time complexity for data parsing increases linearly, the time complexity for database creation increases linearly, and the time complexity for each query increases linearly as well. The reason these all scale linearly is because they are all based on iterating over the set of data. It is probably acceptable for the access policy creation to be slow because that likely wouldn't happen often in practice. However, for a huge system with many users performing many queries, the performance would degrade significantly.

Setup (commands in blue)

Dependencies: Flask, SQL Alchemy

- 1) `pip install flask`
- 2) `pip install flask-sqlalchemy`
- 3) `set FLASK_APP=access.py` this is for Windows, I believe Linux would use `export` instead of `set`

Running (commands in blue)

- 1) `flask initdb` this command parses the JSON file and creates the databases
- 2) `flask run` this launches the application
- 3) go to `http://localhost` in browser and enjoy