Access Control Language

SD3 supports the use of groups letting it support indirection. SD3 also has the built in $ notation, allowing it to support delegation. Role inheritance is also supported in this version of SD3.

The syntax used to write the policy is as follows:

**NOTE:** Options under notation that are in *italics* are optional

Spaces cannot be used for any of the names: group names, usernames, and filenames cannot be separated by spaces.

The three types of permissions are “read”, “write”, “execute”

|  |  |  |  |
| --- | --- | --- | --- |
| **Policy** | **Notation (LHS)** | **Delim** | **Notation (RHS)** |
| **Users** | PKD(“username”, userkey) | :- | ; |
|  | PKD(user, key) | :- | UserKey$PKD(user, key); |
|  | Attr(“username”, “Attribute”) | :- | ; |
| **Groups** | Group( “groupname”, “user-to-add”) | :- | ; |
| **Permissions** | Perms(“user”, “filename”, “Permission*”)* | :- | ; |
| (All user of group have permission for file) | Perms(user, “filename”, “Permission”) | :- | Group(“group”, user); |
| The user is trusted to edit permissions. | Perms(user, resource, op) | :- | PKD("username", key); |
| If the “username” has the permission for resource on RHS then permission of resource is granted to LHS user | Perms(“user”, “resource”, “Permission”) | :- | PKD(“username”, “key”), “username”$perms(user, “resource”, “Permission”); |
|  | Perms(“user”, “resource”, “Permission”) | :- | PKD(“Username”, key), “username”$Attr(user, “attribute”); |
| **Group Hierarchy** | SubGroup(“parentgroup”, “childgroup”) | :- | ; |

**Indirection**

The two main ways to implement indirection is the use of groups and subgroups.

*Perms(user, “This/is/a/file”, “read”) :- Group(“This is a group!”, user);*

The above line of policy grants read access to the file “This/is/a/file” to all users of the group “This is a group!”

*SubGroup(“Parent Group”, “Child Group”) :- ;*

The above line of policy makes “Child group” a subgroup of “Parent group”. This allows all users in the Parent group to have the same access to all files that Child Group has. This allows a large set of permissions to be assigned to the users in the Parent group.

**Delegation:**

The way to implement delegation is to use the Perms policy along with a right hand side argument utilizing the ‘$’ operator.

*Perms(“Dave”, “coolstuff.txt”, “read”) :- PKD(“Sally”, key) “Sally”$Perms(user, “anotherfile.txt”, “write”);*

This policy line is saying : If Sally has write access to anotherfile.txt, then give Dave read permission to coolstuff.txt.

*Perms(“Dave”, “file.txt”, “read”) :- PKD(“Dave”, key) “Dave”$Perms(user, “file.txt”, “write”);*

This policy line Is saying : If Dave has write access to file.txt then give Dave read permission to file.txt.

*Perms(“Dave”, file.txt”, “write”) :- PKD(“Dave”, key) “Dave”$Atrr(user, “Blue eyes”);*

This policy line is saying: If Dave has blue eyes, then he should have write access to file.txt.

*Perms(“Dave”, file.txt”, “write”) :- PKD(“Sally”, key) “Sally”$Atrr(user, “Job”);*

This policy line is saying: If Sally has the attribute job then Dave should have write access to file.txt

**Other Features**

1. Group Inheritance

I have implemented group inheritance in my code. Groups can have child groups. The parent groups will inherit all permissions that a child group has.

*SubGroup(“Managers”, “Employees) :- ;*

This line makes employees a subgroup of managers. All managers will now have manager permissions as well as employee permissions.

*SubGroup(“Employees”, “People”) : - ;*

This line makes people a subgroup of employees. All employees will now have people permissions as well as employee permissions. If both the examples were in the file, then Managers would now also have people permissions.

2. Parameterized Attributes

All users can be assigned attributes, and those attributes can be used to decide if a user should be granted a privilege to a file.

*Attr(“Dave”, “Has Hair”) :- ;*

This assigns the attribute to Dave “Has Hair”. Note that “HasHair” is the same as “Has Hair” when assigning attributes.

*Attr(“Dave”, “Blue eyes”)* :- ;

This assigns Dave the attribute blue eyes. Again “Blue eyes” == “Blueeyes”

**Analysis of example files:**

|  |  |  |
| --- | --- | --- |
| File Name | Proccess Time (ms) | Average Query Time (ms) |
| AccessFileTrustChain.txt | 73.94652 | 0.24871 |
| AccessFileShort.txt | 52.57604 | 0.34727 |
| AccessFileManyUsersManyPermissions.txt | 125.90091 | 0.60571 |
| AccessFileManyEverything.txt | 651.76717 | 1.95537 |

As files become more and more complex, the runtime expectedly increases. From the test results we can see that both the process time and query time increases substantially. AccessFileManyUsersManyPermissions.txt has 60 users each assigned 4 files, and each assigned to groups. This is a total of 240 permissions across all users and files. When we compare this to AccessFileManyEverything.txt with a total off 100 users, each assigned 62 files. This is a total of 6,200 permissions across all users and files. AccessFileShort.txt has a total of 15 permissions. Using these 3 data points we can estimate a growth in the chart below.

With so few data points it is hard to say just by the charts that the program scales linearly, but analyzing the data structure backs up the linear scaling. Whenever permissions are added first the user in the permission is searched for, this is O(n) in number of users. Then the file is searched for which is also O(n) in number of files. If both exist a node is added to each of the Access Object’s tree. This addition is also O(n) because duplicates are not allowed in the tree. This leads to an overall O(n) in the number of permission connections. The same can be said of queries. Querying searches the query subjects tree for the object being queried. From above we know this is O(n). One advantage of the structure is creating chains of trust do not increase runtimes. Since each object has its own access tree you never have to search more than one level to find whether the if statement is true.

I believe it is possible to do better than linear scaling. I think the issue is with the underlaying data structure I set up. Since users, files, and groups are all stored in arrays searching could be improved using a better search algorithm then O(n), or even better a hash map could be used for each object to improve searching. I think query times could also scale better with this type of look up. Instead of child nodes of the trees being stored as arrays, they could also be stored as hash maps to improve the children searches.

Since a large majority of the runtime is preprocessing the file I think that the program as written is suitable for both personal computer usage, possibly small enterprises. Since preprocessing should only have to happen on startup of the system, and query times are relatively small I think it is a reasonable solution. For much larger scale systems, such as cloud storage, the program would need the improvements discussed above.