*Christian Jarani*

*CS 1699: Privacy in Electronic Society*

*Project 2: Access Control Engine*

**TO COMPILE: javac AccessControlEngine.java**

**TO EXECUTE: java AccessControlEngine**

* **Policy files are named init1, init2, init3, and init4.**
* **To select individual policy files, edit line 16 to specify which of the four you would like to run.**
  + **in = new Scanner(new File("init[desired number].txt"));**

**W0**: My access control language will center around a tree data structure of my own creation, with each node in the tree representing a role with some set of permissions. The nodes themselves will be represented as objects containing a set of permissions, a list of users who’ve been granted that respective level of permissions, a list of files (where the role it’s included in meets the minimum permissions required for access), and pointers to the parent and child nodes. Access checks for files only require a best-case **O(log(n))** traversal (*assumes balanced tree structure, hence “best-case”*) of the lists of the role subtree and an **O(n)** traversal of each node’s file list, with a runtime of **O(n\*log(n))** in total. This will ensure scalability for increasingly-large additions of roles, files, and users.

**Indirection** will be supported through role assignment, i.e. adding one user to a role will give them all the permissions of that role which removes the need to delegate them individually. **Delegation** will be handled by hierarchy, where a user of a certain role has the ability to delegate permissions of all roles **directly underneath** them. The ramifications for irresponsible delegation will not be covered in this project, as that falls under the duties of whatever administration is using this system.

In addition to indirection and delegation, I’ve also formatted my data structure to support **role inheritance**, such that the permissions of each role will also include all those attributed to their children. This permission structure is optimal for environments with a hierarchal architecture, such as a business, government, or religious body. Secondly, **attribute intersection** will be supported, meaning users are able to perform multiple roles at the same time (i.e. included in 1+ role’s listed of authorized users) allowing them access to file hierarchies of multiple branches. This access could be permitted as either a window-of-opportunity (i.e. allows certain permissions not normally attributed to a role for a specific amount of time) or permanent access depending on the requirements of the organization using this model. ***The window-of-opportunity option will not be addressed in this project.***

**W1**: The format of the policy file will be separated into three sections: roles, users, and files. Each section will group its contents together as a single string to be delimited by the engine. The order of this listing is strict, such that the first line must define the roles, the second line must define the users, and the third must define the files. An example of the formatting is described below:

Each role will include their identifier, parent role, and their associated set of permissions, separated by a colon. If the parent role is defined as ‘!’, it has no parent i.e. it is the root.

* a:!:r,b:a:rw,c:a:rwa, … [newline]

Each user listing will include their identifier along with the role to which they will be assigned.

* bill:a,chris:f&b, … [newline]

Each file listing will include their identifier, along with the role to which they will be assigned.

* kappa.txt:a,gibber.jpeg:c, …

**Additional Syntax Information**

1. *A list of existing roles isn’t necessary for parsing.*
2. *Each section (roles, users, or files) must be written on a single line.*
3. *Multiple role assignments for a user or file are supported through use of the “&” character.*
   1. *Max of 4 unique role assignments per file/user (can be increased as needed)*

***For tasks W3 -> W5, a graphic representing the tree structure used in the example will be included below:***

**W3**: Indirection is supported in my engine by reading in an init file of variable size, creating the access control tree based off its contents. As described in W1, user listings have an associated role. Roles implicitly dictate the privileges of the users contained within because of the structure of the tree. If I include role listings a, b, c, d, e and f, such that a is the root, b and c are children of a, d and e are children of b, and f is a child of c, with user listings bill:a,chris:f&b,daria:c, bill will acquire all privileges associated with a, chris with f and b, etc. etc. etc. This can be done with any number of users, as roles can accommodate as many users as necessary.

**W4**: Delegation is also dictated implicitly by the tree structure. For instance, take a tree made of 6 nodes: a, b, c, d, e, and f. A is the root, b and c are children of a, d and e are children of b, and f is a child of c. If user x belongs to role b, they may access all of b’s files, as well as d and e’s, and can delegate access to users of d and e to the extent of b’s own privileges. Users of role a have root access, so they are not restricted to delegating access to roles underneath it, but also may delegate the privileges of itself, as well. This ensures that users of a certain privilege can only access higher privilege files if it is allowed by a user of one level of privilege higher than the file they want to access (unless that user is root). If users are a part of multiple roles, the same case is true, but to a larger scale.

**W5**: Role inheritance, too, is also supported implicitly by the tree structure. All roles have access to files within their role and inherit all access of the roles directly beneath them. For instance, role a has access to all files in the system, role b has access to b’s file as well as d and e’s. B does not have access to any file hierarchy on the opposite side of the tree. Following this logic, roles d and e only have access to the files included within their role, as they do not have any children to inherit privilege from.

In the case of attribute intersection, my language supports multiple roles per user up to a maximum of 4. As described in the policy file format described above, attributing a user to multiple roles requires the use of the & character, such that a user entry defined as chris:b&c will have access to all privileges inherent to roles b and c. For larger tree structure, this can ease the pain of sharing files between roles and having to delegate access to only that user and no one else. This architecture requires mindfulness of file assignments in concern to roles, such that multiple role users will not have access to files they shouldn’t.

**W7a**: To measure policy preprocessing, I measured the time it took to build the access control tree from files of varying number of roles, users, and files. Specifically, “varying number” means exponentially scaled numbers of all factors. For example, the base init file will contain 6 roles with 12 users and 12 files, the second will contain 12 roles with 24 users and 24 files, etc. The total execution times displayed in the graph are the averages of 10 rounds of tests. The results can be observed below:

Due to the steadily increasing slope per data point, it can be reason that the execution time scales linearly with proportionally-increasing roles, users, and files. Although the effects appear to be exponential in runtime, the x axis increments exponentially, so an exponential line cancels into linear. For these reasons, the increase in execution time for increasingly large policy files is very small.

**W7b**: To measure query processing time, I measured accesses on each of the policy files I created for the preprocessing measurements. The results can be observed below:

Again, we see a linear increase in query execution time as the size of the preprocessing file increases. Although the increase in execution time is larger for queries than for policy preprocessing, it is still small enough to remain negligible.

**Conclusions**

Due to the highly-scalable nature of this access control architecture, I conclude that this model is appropriate for any scenario requiring role inheritance and attribute intersection. Inheritance and delegation are supported implicitly by the tree structure, so no further policy is necessary in easing the difficulty of these two tasks (reduces the overhead of implementing/execution such policies, as well!). Building off this benefit, this architecture would work well in personal, enterprise, and large-scale environments. Average runtime of access queries is O(log^2(n)) because of the two tree traversals required for the check, one for finding the user and another for finding the file. This runtime is solid enough to hold up in the face of even some of the stauncher adherents of runtime efficiency.

Overall, the system I have implemented should be highly-diverse in terms of its applications, though I’m sure certain operations could be optimized. For instance, filling roles with users and files requires a traversal for each user entered in the system, with a worst-case runtime of O(n\*logn). A potential solution I was considering was to sort users/files into lists before placing them into their respective roles, so only on traversal is required per role to fill it with its respective contents (users/files). But, even without that, my operations scale well in the face of exponentially increasing loads, so I decided to report it only as an idea.