Anna Smith

CS1699 P2 Writeup

W0:

For this project I chose to implement ReBAC, relationship-based access control.

W1:

The policy file should be a JSON file with three main sections: resources, relationships, and permissions. Each of these sections is itself a JSON object. The resources section maps each existing resource to a JSON object defining the resources owner, policy, and delegations. The relationships section maps existing relationship types to an array of participants pairs. For example, one entry may be “parent” : [“Rose, Marie”, “Rose, Terri”]. Relationships that go both ways must be stated both ways. For example, “sibling”: [“Sarah, Tim”, “Tim, Sarah”] will not work without both relationships stated. The Permissions section defines existing permissions, mapping a character representation to a string describing the permission. For example, “r” : “read” means that a permission called “read” exists, and it can be referred to by “r” when defining policies.

The resources section is the most complex of the sections. As stated above, it maps a resource to its owner, its policy, and its delegations. The policy is what actually expresses a ReBAC policy. The syntax of the policy itself is defined as the disjunction of relationship statements and delegation statements. A relationship statement expresses a relationship to the owner and a set of permissions, meaning that these permissions are to be granted to anyone with the stated relationship to the owner. A delegation statement expresses a relationship to the owner, indicating that individuals with this relationship to the owner are able to delegate a subset of their own permissions. The delegations themselves are defined in the delegations section. The syntax used in a policy is defined as follows:

Relationship Syntax

* a = represents the owner of the object
* <i>% = Any person c such that owner a is related to another person b with relationship i and b is related to c as described in formula %. Ex. <superior>a is anyone who is a's superior. Ex. <parent><sibling>a is anyone who is a's parent's sibling.
* <-rel> = the opposite of a given relationship. Ex. <-parent>a is a's child
* T = represents everyone
* !% = negation symbol. refers to anyone who doesn't fulfill the formula %

Ex. !<parent><parent>a refers to anyone who isn't the grandparent of the owner

Statement Syntax

* R(permissionTypes) = Relationship statement. Anyone with the relationship R to the owner is granted the permissions listed in permissionTypes. Permission types are referred to with a single character and should be defined in the Permissions section of the policy file.

Ex. !<parent><friend>a(rwx) means that anyone who is not the owner's parent's friend has accesses corresponding to r, w, and x.

Ex. T(r) means that everyone has access corresponding to r.

* $(R) = Delegation statement. Anyone related to the owner via relationship R can delegate a subset of the permission types that they have.

Ex. $(<parent>a) means a's parent is able to delegate a subset of their permissions

Ex. $(T) means anyone can delegate their permissions

Policy Syntax

* | = disjunction of two statements

Ex. a(rwx)|<parent>a(rw)|!<sibling>a(r)|$(<parent><parent>a, <parent>a) means that

-the owner has permissions r,w,x

- parents of the owner can read and write

- anyone who isnt the sibling of a can r

- owner's parents can delegate a subset of their permissions to the owner's grandparents

So, relationships are used to create relationship statements and delegation statements, and these statements are then joined via disjunction to create a policy for a given object.

The actual delegations for a resource are defined the delegations section that exists per resource, a JSON object that maps a delegator to a delegation. The syntax of a delegation is almost the exact syntax as a relationship statement; the only difference is “d” is used instead of “a”, to indicate that the delegations grant access via a relationship to the delegator themselves, not the owner of the resource. For example, "Erin”: {"delegates" : "<friend>d(r)” } means that Erin is attempting to delegate her read permission to her friends. A delegation is only valid if there is a delegation statement allowing the delegator to delegate, and the delegator is granting a subset of their own permissions.

As a side note, this syntax is not able to explicitly deny any permissions. Permissions are denied by default, and policies and delegationscan only grant permissions. So, although the policy !a(r) means that everyone but the owner is given read permission, it does NOT mean that the owner cannot be given read permission.

C2:

The main driver of the program is driver.py. It can be run with the command “python3 driver.py filename”, where filename is the name of the policy file to be used. The two sections of the driver program are the preprocessing of the policy file and the query loop (lines 5-15).

The preprocess function kicks off the 4 parts of preprocessing the policy file: retrieving the permissions dict, building a social network graph out of the relationships dict, parsing the policy statement for each resource, and processing the delegations for each resource (lines 42-67).

* The permissions dict is simply retrieved from the JSON file as is (line 52).
* The buildSocialNetwork method builds the social network graph out of the relationships dict (lines 191-231). The network is represented with the class Graph defined in graph.py. Each person in the graph is represented as a Node, and each relationship is represented as an edge.
* Lines 58-62 in the preprocess method call processPolicy for each resource’s policy. processPolicy takes a string representation of a policy and turns it into a policy object, defined by the Policy class in statements.py (lines 235-252). It does this by splitting the string policy into statements and calling parseDelegationStatement if it’s a delegation statement, or parseRelationshipStatement if it’s a relationship statement. The parseDelegationStatement method takes a delegation statement and returns a DelegationStatement object, defined by class DelegationStatement in statements.py (lines 312-336). The parseRelationshipStatement method takes a relationship statement and returns a RelationshipStatement object, defined by class RelationshipStatement in statements.py (lines 257-308). The policy object returned from processPolicy is constructed from the DelegationStatement and RelationshipStatement objects returned from these calls.
* processDelegations retrieves the delegations from each resource, and checks whether each is a valid delegation. If it is a valid delegation, it is turned into Delegation object, defined in statements.py, and appended to the policy object (lines 74-121). A delegation is valid if there is a delegation statement that allows the delegator to delegate, and the delegator is delegating a subset of their own permissions. So, the delegations are cross-referenced with DelegationStatements in the policy to verify ability to delegate, and with RelationshipStatements in the policy to verify that they are delegating permissions they actually possess. After processing the delegations for a resource, the DelegationStatements in the policy are ignored, as only the RelationshipStatements and Delegations are used to evaluate access.

After the preprocessing is complete, the driver program moves on to the query loop by calling the queryLoop method. Each query consists of a resource, an accessor, and a permission. The user is asked to enter these one by one. The query is then evaluated, printing either “Access Granted” or “Access Denied” to the user (lines 18-38). The query is evaluated using the hasAccess method, which takes a resource, an accessor, a permission, the relevant social Network, the resources dict, and the nodes in the social network graph, and returns either True or False. The hasAccess method iterates through all the objects in the Policy object for the given resource to see if any of them grant the desired permission to the accessor (lines 130-160). If a given object is a relationship statement, the relatedVia method is called to evaluate the relationship between the owner and the accessor. If a given object is a delegation, the relatedVia method is called to evaluate the relationship between the delegator and the accessor. Delegation statements are ignored. If any of these relatedVia calls return True, that means that that part of the policy does grant the desired permission to the accessor, so has Access returns true. If there is no relationship statement or delegation granting the desired access, has Access returns False.

There is one known bug that I was not able to fix: if an accessor is not connected to anyone, they will not be given access through policies like !<parent>a, because there is no way to evaluate the relationship between the owner and the accessor.

W3:

Indirection can easily be implemented in this ReBAC language, as each relationship statement can grant permissions to any number of people. For example, <friend>a(r) would grant read access to all friends of the owner. Depending on the number of friends the owner has, this could be many people. <-parent>a(rw)|<-parent><-parent>a(rw) grants read access to all children and grandchildren of the owner, which again could be many people. So, it is simple to grant permissions to many people at once.

W4:

Delegation where the owner is delegator is implicitly supported via relationship statements. A policy containing the statement <friend>(r)a means that the owner has implicitly delegated read permission to their friends. A policy containing the statement <-parent>a(r) delegates read permissions to the children of the owner.

W5:

This language also supports delegation by the accessor of an object. If an individual has access to a resource, they can delegate a subset of their permissions to others if there exists a delegation statement that allows them to delegate. If a resource’s policy is a(rwx)|<friend>a(rw)|$(T), anyone can delegate a subset of their own permissions. If Andrea is a friend of the owner, and there is a delegation “Andrea” : {“delegates”: “<friend>d(r)”}, Andrea’s friends gain read access. If a resource’s policy is a(rw)|<parent>a(rw)|$(<parent>a), only the owner’s parents can delegate. If John is the parent of the owner and there is a delegation “John” : {“delegates”: “<sibling>d(rw)”}, siblings of John gain read and write access.

C6:

The 5 experimental policy files I created are in t1.json, t2.json, t3.json, t4.json, and t5.json, in order of increasing complexity. Here, I am defining complexity as the number of relationship statements, the number of delegation statements, and the number of delegations. So, as t1.json has one of each, it is complexity = 1. t2.json has complexity = 2, t3.json has complexity = 4, t4.json has complexity = 6, and t5.json has complexity = 8.

Clearly, this is an imperfect way of measuring complexity, but it is simple to understand. The complexity of the social network certainly also affects the time taken to preprocess the policy file and to respond to queries, but again, for simplicity’s sake, I focused on the complexity of the policies themselves while keeping the social network constant.

timer.py calculates the average times to preprocess and respond to queries for each of the 5 experimental policies. To run the test, use the command “python3 timer.py”, assuming all 5 experimental policy files are in the same folder as timer.py. For each policy file, it runs the preprocess function the number of times indicated in the variable trials. Then, it calculates the average time taken to complete the preprocess method (lines 10-18). After this it runs the hardcoded query the number of times indicated in the variable trials. It then calculates the average time taken to complete this query (lines 21-27). Both the average preprocessing time and the average query time are printed for each policy file. The hardcoded query was chosen so that the response from hasAccess will be False for each of the policy files. This way, hasAccess has to iterate through each of the relationship statements and delegations in the relevant policy; increasing the complexity of the policy would impact the query speed the most in this case.

W7:

The following graphs show the results of running timer.py:

These results imply that, using the above definition of complexity, the time taken to perform preprocessing and the time taken to respond to a query both increase roughly linearly with complexity. These results also show that the preprocessing time dwarfs the query response time. These results are far from conclusive, as there are only 5 data points, and it is unclear to what degree individual factors such as number of delegation statements, number of delegations, and relationship statements impact the runtime. So, I am not able to conclude that the runtime of the preprocessing or the querying is in fact linear with respect to any of these individual variables without closely inspecting the code and running more comprehensive tests.

In terms of the access control language itself, this type of language would probably best be used in applications that would benefit from delegation and indirection based on human relationships, such as an application for sharing photos and photo albums among friends and family, or health records systems.