CS 1699: Privacy in the Electronic Society
Project 2 – Access Control Policies

Jonathan Dyer

April 6, 2018

Contents

Task W0 **2**
- **RT$_0$**: Role-based Trust management with attributes . . . 2

Task W1 **3**
- Overview of Syntax and Policy File Format . . . 3
  - Entities . . . 4
- Data . . . 4
  - Objects . . . 4
  - Roles . . . 5
  - Subjects . . . 5
- Policies . . . 6
  - Hierarchy . . . 6
  - Delegation . . . 6
  - Inference . . . 6

Task C2 **7**

Task W3 **7**
- Indirection . . . 7

Task W4 **7**

Task W5 **7**

Task W7 **7**

References **8**

Task W0

$\mathbf{RT}_0$: Role-based Trust management with attributes

The access-control language selected for this project is a simplified variant of a family of expressive languages known as **RT**, which is short for ***R**ole-based **T**rust-management*. In spite of the name, the **RT** family is actually an extension of role-based access control (RBAC) known as *attribute*-based access control (ABAC). It fulfills all of the desirable traits mentioned in [1], including delegation of attribute authority, inference of attributes, and more. By allowing explicit subject abstraction and specific attribute assignment to those subjects, the system provides the same functionality that roles do via selection by attributes and intersection of attributes along with even greater flexibility and expressiveness. The language utilized here is based on a combination of $\mathbf{RT}_0$ and $\mathbf{RT}_2$, and functions as follows:

- The primary structures in this framework are *entities* (or principals), *subjects* (or users), *roles* (which contain rights or permissions), and *objects* (or resources).
- **Entities** are simply the organizations or systems that issue credentials (i.e. assign roles to users). By explicitly abstracting entities, the **RT** framework allows for localized roles, as well as more extensive delegation.
- **Subjects** are the users or agents in the system. They may have one or more roles that provide them with permissions (or authorized actions) for accessing objects in the specified way.
- **Roles** are a convenient way of grouping permissions that may be assigned to subjects. Roles may be hierarchical–if one role dominates another, then it has every permission the other role has. This helps reduce the number of roles and relationships that have to be dealt with in the system. Role assignment may be achieved by specifying subject attributes.
- **Objects** are the elements whose access is being controlled by this entire schema. They may be grouped by attribute as well, allowing flexible and powerful specification of permissions.
- Each structure accepts descriptive *attributes* that enable powerful specification of policies based on any desired combination of subject, role, or object attributes. This, in conjunction with the outline above, facilitates the following features (among others):
    1. **Indirection**: Because we can specify policies according to attributes, we may assign i) a set of permissions (i.e. a role or multiple roles), to ii) a set of users, over iii) a set of objects, as long as those sets can be well-defined or uniquely specified (i.e. all elements in the desired set must share an attribute that specifies exactly that set).
    2. **Delegation**: Different entities that assign roles (within their domain) may easily defer to the authority of another entity for checking role membership. That is to say, if organization A defines role r_1, it is simple to write a policy that includes into r_1 all members of role r_2 from some other organization B as well. This is also referred to as 'delegation of attribute authority', meaning that if B says that some subject has attribute r_2, then A says it has attribute r_1 (here speaking of roles as attributes of users). Details of how this is done are given in the next section.
    3. **Role Hierarchy**: Roles may inherit permissions from other roles given by the same entity. This is a variant of delegation, in a sense allowing an organization to delegate authority over some role attribute to that same organization under another role attribute. Thus, role management is simplified and duplication is reduced.
    4. **Logical Objects**: Inspired by $\mathbf{RT}_2$, the current variation of this framework also supports collections of objects, known as Object Groups, which allows addition by attribute and assignment of permissions respecting objects *en masse*. In other words, it is possible to define a role with an access mode and an entire group of logically-related objects (via attributes) rather than just one object at a time.
    5. **Attribute Inference**: The **RT** framework is flexible enough to allow specification of inference rules. This essentially provides a conditional ('if-then') logic that allows you to assert some property of a subject, object, or role as long as it has some other property.
    6. **Attribute Intersection**: Support of multiple attributes on any given element in the system also allows specification of policies by intersection of attributes. For instance, it is possible to specify an action that is allowed only for users who are in the intersection of multiple roles (i.e. users that have each of those roles). This feature is **not implemented** in the current project.

It is important to note that there are many other extensions and variations in the **RT** language family, and the current iteration was chosen as a convenient balance between expressiveness and feasibility for the purposes of the current project. Many other features are possible within the **RT** framework, such as parameterization of attributes, threshold policies, and more.

Further, it is important to note that the original specification of **RT** also included details regarding the issue of *common vocabularies* across entities. This issue of vocabularies or namespaces is dealt with via *application domain specification documents (ADSDs)*, but was ommitted here for brevity's sake, and because it is not relevant to the ideas being explored. For more information however, and generally for a great amount of detail regarding the **RT** family of languages, see [1] and [2].

Task W1

Overview of Syntax and Policy File Format

The syntax chosen to write policy files for the above language is straightforward and designed for maximal ease and clarity. It is encoded in XML, and comprises two primary sections in every entity for which policies are being defined: 1) Data, and 2) Policies.

Data is where all elements of the entity are specified, and includes the following:

- A full definition of any objects in the entity, including Logical Objects (object groups) and any attributes associated with them.
- A definition of all roles and their corresponding permissions, including all objects/object groups those roles affect. Any role hierarchy will be defined in the policy section.
- Specification of all subjects or users given credentials (i.e. assigned roles) in the entity.

Policies is the section where any of the extra features or relationships are defined. Although many of the basic access policies are implicitly encoded in the Data section, more advanced relationships are expressed here, including:

- Delegation of authority.
- Role hierarchy definition.
- Access that relies on attribute intersection (not discussed in this document).
- Attribute inference or other complex relationships.

Thus an overall outline of the policy file may look like this:

```
<?xml version="1.0"?>
<!-- The <root> element simply contains everything and exists for parsing convenience -->
<root>
    <entity name="Entity 1">
        <data>
            <objects>
              ...       <!-- All object groups and resources go here -->
            </objects>
            <roles>
              ...       <!-- All roles and permissions go here -->
            </roles>
            <subects>
              ...       <!-- All subjects and their attributes go here -->
            </subjects>
        </data>
        <policies>
          ...           <!-- All specific policies go here -->
        </policies>
    </entity>
    <entity name="Entity 2">
      ...               <!-- Repeat for as many entities as you're defining -->
    </entity>
</root>
```

In keeping with generally accepted XML style, all core and necessary features of an element are contained in sub-elements, while optional or incidental data, including labels (i.e. attributes) are contained as XML attributes in the opening tag.

It is important to keep in mind that the description here is for a basic outline of a policy file, including implementation of features 1-5 above. Further extensions and options are possible and may be made available at a later time, but this describes the minimum syntax necessary to write an access control policy file with these features in **RT** using an XML encoding.

Note: Throughout this document the term "administrator" will be used to refer to the writer of the policy file.

Entities

The only thing to know about the `<entity>` element is that it has a single attribute called 'name' by which you may indicate the name of the entity. In order to utilize delegation, relevant entities *must* be named. Note that this implementation does not currently support hierarchical entities. Otherwise, each section (Data + Policies) necessary for an entity's fully-described access control is explained forthwith.

Data

The `<data>` tag has no attributes, so we will move directly into considering objects, roles, and subjects.

Objects

This section is straightworward to write: simply define any objects in your system, and assign them to groups if desired. The `<objects>` tag itself takes no attributes, but rather serves to enclose the section. Two types of elements can be contained in this section:

1. Object Groups are denoted by the `<objectGroup>` tag and must have a name, and otherwise have no requirements.
2. Individual objects, denoted by the `<object>` tag, must have a name and otherwise have no requirements. They may also be assigned to one or more groups, specified as sub-elements with `<group>`.

Our first feature is introduced in this section, namely *Logical Objects* (called 'Object Groups' from now on). This refers to the ability to specify objects in one of two ways: individually by name, or as a collection via a formalized Object Group such as 'Files' or 'Printers'. Here I give an example of how to represent both in the XML encoding schema. Usage of (1) and (2) in access policies is covered in *Roles*. Detailed examples of Logical Objects will be provided later on in section **W5**.

The snippet of code below shows one Object Group called 'Files' and three objects, two of which are assigned the Files group and one of which is not assigned any group.

```
...
<objects>
    <objectGroup description="This group is for regular old files">
        <name>Files</name>
    </objectGroup>
    <object description="This object is in the group 'Files'">
        <name>users_file.txt</name>
        <group>Files</group>
    </object>
    <object description="This object is also in the group 'Files'">
        <group>Files</group>
        <name>passwords_file.txt</name>
    </object>
    <object description="This object has no group">
        <name>LabPrinter</name>
    </object>
</objects>
...
```

- Note that each object or Object Group may have one or more attributes given in the opening tag (such as 'description'). These are optional and defined by the administrator, and are not used in the current implementation of **RT** but may be supported in some forthcoming feature (such as printing descriptions of structures in the policy file). They do not affect functionality and may be safely ignored.

- Significantly, we may assign an object to as many groups as we wish. In future releases, this would make possible the specification of objects by union/intersection of Object Groups, in what is known as 'Attribute Intersection'.
- Finally notice that order of sub-elements does not matter – `group` may come before or after `name`.

Roles

The roles section is only slightly more complex, since all advanced role manipulation takes place in the Policies section. Of special note below, we *must* designate a name for every role, else it will not be useful. The `<objects>` tag itself requires no attributes.

Also, because the *types* of access or permission tend to be far fewer than the number of objects being accessed, we will always define permissions according to type, followed by a list of objects that permission applies to, rather than entering every object and associated permissions individually. This saves a great deal of effort on the part of the administrator.

Our next feature, *Indirection*, is best noted here, since it has primarily to do with assigning sets of permissions. These sets are abstracted as roles, and make it easy to lump permissions that are commonly assigned together.

```
...
<roles>
    <role description="Everyone gets put into this role">
        <name>User</name>
        <permission description="Read rights for all Users">
            <type>Read</type>
            <target type="object">users_file.txt</target>
        </permission>
        <permission>
            <type>Write</type>
            <target type="object">users_file.txt</target>
        </permission>
    </role>             <!-- End role 'Member' -->
    <role description="This is only for superusers">
        <name>Admin</name>
        <permission description="Read all Files and objects for Admins">
            <type>Read</type>
            <target type="objectGroup">Files</target>   <!-- I can assign a group at once -->
            <target type="object">LabPrinter</target>   <!-- or just a single object -->
        </permission>
        <permission description="Write to all Files and objects for Admins">
            <type>Write</type>
            <target type="objectGroup">Files</target>
            <target type="object">LabPrinter</target>
        </permission>
    </role>             <!-- End role 'Admin' -->
</roles>
...
```

- All permissions must be described within the context of some role, and roles can have as many permissions as desired.
- The tag `<target>` defines what resource(s) the permission is referring to, and a given right may be assigned for an indefinite number of targets.
- The '`type=`' attribute is mandatory metadata that specifies the type of target being accessed; currently only objects and Object Groups are supported.

Subjects

The subjects section is the final sub-section of **Data**, and contains information about all users or actors in the system. Here they are assigned roles which give them the permissions associated with those roles. This also supports *Indirection*, by allowing users to be associated with a large set of permissions at once (in the form of a role), perhaps even pertaining to a large set of objects at once (in the form of Object Groups)!

Of note below is that every subject *must* have a name, but not every subject must have a role (although users without a role will have no permissions, it is possible that such a circumstance is necessary). Otherwise this section is written in a natural and simple way.

The `<subjects>` tag itself requires no attributes, but simply encloses the section.

This section of code shows the user 'Alice' being defined and assigned the role 'Admin'.

```
...
<subjects>
    <subject>
        <name>Alice</name>
        <role>Admin</role>
    </subject>
</subjects>
...
```

Policies

Now for the section on writing more advanced rules. Most policies can be written into the **Data** section above. Each feature that has not already been covered – *Delegation, Role Hierarchy,* and *Attribute Inference* – is briefly presented here, and all features will receive in-depth treatment in sections **W3** through **W5**.

The `<policies>` tag requires no attributes, and there can be as few or as many sub-elements as desired.

Hierarchy

This is the simplest policy to define, and takes the form of an 'if-then' statement. All role hierarchies can be described this way: "*If* some user has the role r_1, *then* that user also has the role r_2". This is another way of saying that "Role r_1 *dominates* role r_2". The format is simple:

```
...
<policies>
    ...
    <hierarchy>
        <if>Role_1</if>
        <then>Role_2</then>
    </hierarchy>
    ...
</policies>
```

This will become useful for specifying specialized roles without repeating a lot of work. See **W5** for more details.

Delegation

Delegation is straightforward as well, although it requires more thought in order to write a useful policy. In **RT**, because all access rights are assigned via roles, delegation takes the form of "Give some subject a role in one entity as long as it has some role in this other entity". Again policy may take the form of a conditional: "*If* some user has the role r_1 in entity e_1, *then* that user also has the role r_2 in entity e_2". Note that if the roles r_1 and r_2 are the same role r, then this becomes a delegation from e_2 to e_1 of authority over r. Conversely if the entities are the same entity e, then this becomes a case of role hierarchy.

```
...
<policies>
    ...
    <delegation>
        <if>Role_1</if>          <!-- if this role   -->
        <from>Entity_1</from>    <!-- in this entity -->
        <then>Role_2</then>      <!-- then this role -->
        <to>Entity_2</to>        <!-- in this entity -->
    </delegation>
    ...
</policies>
```

The order of the elements is not important, but the presentation above may provide the best readability. Note that the `<to>` value must be the entity within which the policy is defined. See **W4** for more details.

Inference

Finally the attribute inference policy. As the only structures that currently support attributes are objects (in the form of Object Groups), this policy will have a fairly straightforward syntax as well, also in the form of a

conditional: "*If* some object is in group g_1, *then* that object is also in the group g_2". See the following snippet for the structure.

```
...
<policies>
    ...
    <inference>
        <if>Group_1</if>
        <then>Group_2</then>
    </inference>
    ...
</policies>
```

Essentially what we have described here is a type of Object Group Hierarchy, which allows inference of membership in one group to membership in another, thus the syntax is identical to the role hierarchy policies. More detailed examples are found in section **W5**.

Task C2

See the accompanying program `parser.py` for full details of the enforcement engine. It provides responses to standard access queries to answer the question "Can user u access file f with privilege p?" It also allows greater exploration of the consequences of the specified access policy file, all via a convenient interactive menu. This enables someone to discover which subjects have which roles in an entity, or what the permissions of each role are, among other information.

Task W3

Indirection

Here we provide more detailed and thorough examples of the indirection feature. As previously mentioned, indirection refers to the ability to assign large sets of permissions at once. This is achieved primarily through the use of roles, although object groups can be of some help here as well. Roles are interpositioned between subjects and their rights regarding objects, and it is this position which enables the indirection. The use of roles is quite simple, as the following two samples shall illustrate.

Our first example is of the most basic usage. We first define a few roles in terms of their permissions or privileges over several individual objects (assumed to be previously defined), and then assign those roles variously to subjects as we wish.

```
asdflkja;slkfda;lsdkf;aslkdjf
```

Task W4

Explain with 2 examples how to implement *delegation*.

Task W5

Explain with examples how to implement *2 other features*.

Task W7

Filler.

References

[1] Ninghui Li, John C. Mitchell, and William H. Winsborough. "Design of a Role-Based Trust Management Framework". In: *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2002, pp. 114–130.

[2] Ninghui Li, William H Winsborough, and John C Mitchell. "Distributed credential chain discovery in trust management". In: *Journal of Computer Security* 11.1 (2003), pp. 35–86.