I-RBAC 2000:
A DYNAMIC ROLE TRANSLATION MODEL FOR SECURE INTEROPERABILITY

BY

APU CHANDRASEN KAPADIA

B.S., University of Illinois at Urbana-Champaign, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

ABSTRACT

The secure interaction between two or more administrative domains is a major concern. In this thesis we examine the issues of secure interoperability between two security domains operating under the Role Based Access Control (RBAC) Model. We propose a model that quickly establishes a flexible policy for dynamic role translation. The role hierarchies of the local and foreign domains can be manipulated through the Role Editor which is used to set up associations between these hierarchies. These associations result in a combined partial ordering of the role hierarchies, which can be used to make meaningful access control decisions for secure interoperability.

To demonstrate the efficacy of this approach, we integrated our policy framework into *Seraphim*, a componentized framework for dynamic security polices. Seraphim is used while making access control decisions in an *active network*. Our policy framework translates the credentials of foreign agents into credentials that are meaningful in the local domain.

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Roy Campbell for his help, support, and guidance. I also thank Prof. M. Dennis Mickunas and my colleagues at SRG. I thank Jalal Al-Muhtadi for help with developing the I-RBAC 2000 model. I would also like to thank the members of the Active Networking Group for their help and patience while I modified their work on Seraphim.

I am also grateful to the Department of Defence[1] for financial support.

[1]DOD grant MDA-904-98-C-A895

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

We use the terms *domain* and *security domain* to refer to an *administrative domain*, which is defined as "a collection of hosts and routers, and the interconnecting network(s), managed by a single administrative authority [8]." This single administrative authority will include a *security officer*. We assume that the domains operate under the Role Based Access Control (RBAC) model [9]. In RBAC, a user is assigned to a *role* that indicates the user's function in his or her organization.

Consider the scenario when two security domains, A and B, desire to interoperate securely. In order to do this, A and B need to establish a secure context. We define a *secure context* to be "a secure session between two entities subject to a domain security policy." To establish a secure context both domains need to agree on a security policy. Both domains can revert to a default security policy that provides a basic level of security. However, such naïve approaches are static and do not offer flexibility to security aware applications and domains. For example, if a client object C, in domain A wants to establish a secure context with a target object T, in domain B, it must rely on the

underlying security mechanism to establish the secure context. For a higher degree of flexibility, both the client and target objects C and T, should be aware of each other's identities. This occurs naturally within a single domain. However since C and T are in different domains, their identities are generally unknown to each other. In RBAC, if the client object C has the foreign role of *Professor*, and if the target object T usually interacts with client objects with the local role *Manager*, how would each domain determine the relevance of the foreign roles? This is the problem that we will address.

1.1 Proposed framework

To solve this problem, we propose a policy framework that facilitates the secure interoperability between two or more domains. The policy framework works with a set of associations between the local and foreign role hierarchies. These associations form a combined hierarchy that is partially ordered. This combined partial ordering is used to attain the level of flexibility desired. Foreign roles can now be translated into local roles, which are understandable to local entities. At the very least, it provides the default role translation. An example of the minimal translation is when all foreign roles are treated as a single Guest role in the local domain. At the other extreme, it allows the security officer to specify a highly explicit role mapping (we call this a *complete policy*, which will be explained later). An example of a more explicit mapping (we call this a *partial mapping*) is when the security officer specifies that a Professor from domain A is equivalent to a Manager in domain B.

Once these associations have been set up, all foreign roles are dynamically translated into local roles. Applications can now make meaningful access control decisions based on the translated roles. These associations are managed through the *Role Editor*, which is at the heart of our policy framework and is the security officer's tool for secure interoperability. Our model is named the Interoperable Role Based Access Control Model 2000 (I-RBAC 2000).

1.2 Deployment in an active network

To show that our policy framework is indeed flexible and useful, we have added it to *Seraphim*, a componentized framework for dynamic security polices [6]. *Seraphim* was developed to secure *active networks*. For example, an active *capsule* or *agent* from a foreign domain will have credentials issued by its original domain. However, for secure interoperability, the local domain needs to translate the foreign credentials into local credentials. This is achieved by our policy framework that translates the foreign role into a local role. New credentials are generated based on the translated roles. Hence, the credentials are effectively translated into credentials understood by the local domain.

CHAPTER 2

DYNAMIC ROLE TRANSLATION

In this chapter, we discuss our *role translation model*. This translation is achieved by adding associations originating from the foreign role hierarchy into the local role hierarchy. These associations allow us to translate foreign roles into local roles. Section 2.1 describes our policy framework formally. We explain the notion of *transitive* and *non-transitive* associations and their importance. A combination of such associations can constitute different security polices, which is described in Section 2.1.1. In Section 2.2 we address security issues with respect to our model.

2.1 Policy framework

Consider the simple role hierarchies H_0 and H_1 for domains D_0 and D_1 respectively, described in Figure 2.1. An arrow directed from a role x to a role y means that x is the parent of y and is hence higher than y in the hierarchy. Although the structures of the role hierarchies are similar, they differ in their semantics. If a client object from a foreign

domain, with the role *Manager*, wants to interoperate with an application in the local domain, that usually allows only local *Professor* roles, the application must be able to translate the foreign *Manager* role into something meaningful. We also observe that both domains have the *Guest* role. If foreign roles are not understood, one can define a simple policy framework to treat all foreign roles to be equivalent to the local *Guest* role. However, this kind of an approach is not very flexible, since all foreign roles are considered to be of only one type, i.e., the local *Guest* role.

Our policy framework creates a combined partial ordering by adding a set of associations between the two role hierarchies. By doing so, one can easily manipulate the levels of access for specific foreign roles.

To formalize this, let R_0 denote the set of roles in the local domain D_0. Let R_1 denote the set of roles in some foreign domain D_1. Then the role hierarchies H_0 and H_1 are partial orderings as shown in Figure 2.1 (solid arrows only).

Definition 2.1.1 *If x and y are roles, we define $x > y$ to mean that x is higher than y in the hierarchy, or, x is the* ancestor *of y.*

Role-names with subscripts, for example $Manager_{R_0}$, will read as *Manager* from R_0.

Definition 2.1.2 *Let R_1R_0 denote the set of associations from R_1 to R_0, where R_1 and R_0 are role hierarchies. We define an association directed from role x_{R_1} to role y_{R_0} to be an ordered pair (x_{R_1}, y_{R_0}) such that $(x_{R_1}, y_{R_0}) \in R_1R_0 \subseteq R_1 \times R_0$. We also denote (x_{R_1}, y_{R_0}) as $x_{R_1} \mapsto y_{R_0}$. Such an association implies that if y_{R_0} is the ancestor of a*

particular role in R_0 *then* x_{R_1} *is also an ancestor to that role. Formally,* $\forall z_{R_0}, y_{R_0} > z_{R_0}$ *implies* $x_{R_1} > z_{R_0}$.

Once these associations are defined, we obtain a dynamic role translation model for secure interoperability.

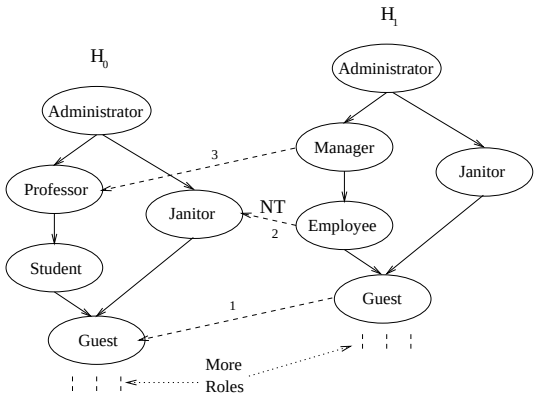

Figure 2.1 Associations between hierarchies - dotted lines

For example, in Figure 2.1 we have the association from $Manager_{R_1}$ to $Professor_{R_0}$ (labeled as 3). This implies that $Manager_{R_1}$ from the foreign domain D_1 will be translated to $Professor_{R_0}$ in the local domain D_0. We will use the following notation to symbolize an association: $Manager_{R_1} \mapsto Professor_{R_0}$. We can also write this as $(Manager, Professor) \in R_1 R_0$.

We define two kinds of associations: *transitive* and *non-transitive* associations.

Transitive Associations: In Figure 2.1 we can see the association $Guest_{R_1} \mapsto Guest_{R_0}$ (labeled as 1). Hence $Guest_{R_1}$ will be translated to $Guest_{R_0}$. For a transitive association, this implies that all the ancestors of $Guest_{R_1}$ will map to the $Guest_{R_0}$.

Definition 2.1.3 *Consider the association* $x_{R_1} \mapsto a_{R_0}$. *If this association is a* transitive association *then* $\forall y \in R_1$, *if* $y_{R_1} > x_{R_1}$, *then* $y_{R_1} > a_{R_0}$.

We can say that y_{R_1} is the *ancestor* of a_{R_0} since y_{R_1} implicitly maps to a_{R_0}. This is a transitive property in which roles inherit the associations of other roles that are lower in the hierarchy. Transitivity is also followed in the local hierarchy. Therefore, $\forall b \in R_0$, such that $a_{R_0} > b_{R_0}$, if $x_{R_1} \mapsto a_{R_0}$ then $x_{R_1} > b_{R_0}$. Hence these associations are called transitive associations.

Non-transitive Associations: The security officer may specifically want to grant access to a particular foreign role and *only* that foreign role. At the same time, the security officer does not have the power to alter the foreign hierarchy. For example, in Figure 2.1, the security officer may want to specifically translate $Employee_{R_1}$ to the $Janitor_{R_0}$, and deny $Administrator_{R_1}$ and $Manager_{R_1}$ from inheriting this association. If the security officer had the power to alter the foreign hierarchy, the officer could semantically achieve this by altering the foreign role hierarchy. But the security officer of the local domain does *not* have the ability or power to do so. Hence, we introduce the concept of a non-transitive association. In Figure 2.1, we can see a such an association between $Employee_{R_1}$ and $Janitor_{R_0}$. We will use the following notation to denote such a mapping: $Employee_{R_1} \mapsto_{NT} Janitor_{R_0}$ or $(Employee, Janitor)_{NT} \in R_1 R_0$.

Definition 2.1.4 *An association* $x_{R_1} \mapsto_{NT} a_{R_0}$ *is non-transitive if it is an association that does not possess the property of a transitive association. In other words,* $y_{R_1} > x_{R_1}$ *does not imply that* $y_{R_1} > a_{R_0}$. *However,* $a_{R_0} > b_{R_0}$ *does imply* $x_{R_1} > b_{R_0}$.

2.1.1 Role translation policies

A set of transitive and non-transitive associations between the foreign and local hierarchies can be used to create a combined partial ordering and define a security policy. Such policies can be put into three categories:

Default Policy: This policy involves setting up a minimal number of associations between a set of roles in the foreign hierarchy and $Guest_{R_0}$. These associations will allow a particular set of foreign roles to interoperate at the *default* level of security. In Figure 2.1 we can see the association $Guest_{R_1} \mapsto Guest_{R_0}$. This corresponds to a default policy. $\forall x \in R_1$, if $x_{R_1} > Guest_{R_1}$, then $x_{R_1} > Guest_{R_0}$.

This scheme is the easiest to set up, but is also the least flexible. This is because all foreign principals are treated as the same role, i.e., $Guest_{R_0}$. However, this scheme is important since it allows the basic level of interoperability and can be used in conjunction with a partial policy, which is described next.

Complete Policy: In this policy, the security officer specifically maps every foreign role to a set of local roles. This policy is an extreme case that illustrates the maximum amount of flexibility in which the security officer can make the mappings explicit for *each* foreign role. An effective way to do this would be to make non-transitive associations from every role in R_1 to a subset of roles in R_0.

Partial Policy: This policy illustrates the true flexibility of our dynamic role translation model. A mapping is partial if it is not a complete policy and when there are one or more associations, usually in addition to the default policy. In such partial hierarchies,

foreign roles without explicit associations still have logical associations by means of the partial hierarchy.

The associations labeled 1, 2 and 3 in Figure 2.1 illustrate this policy. We can see that $Manager_{R_1}$ is higher than $Professor_{R_0}$ ($Manager_{R_1} > Professor_{R_0}$) since we have the association $Manager_{R_1} \mapsto Professor_{R_0}$. Hence, $Manager_{R_1}$ will be translated into $Professor_{R_0}$. Similarly, $Administrator_{R_1} > Professor_{R_0}$ since $Administrator_{R_1} > Manager_{R_1}$. Hence, $Administrator_{R_1}$ will be translated into $Professor_{R_0}$.

For example, applications in D_0 that usually grant access to $Student_{R_0}$ (Figure 2.1) also grant access to the foreign $Manager_{R_1}$ role since $Manager_{R_1} > Student_{R_0}$. It is important to note that $Employee_{R_1}$ will be treated as $\{Guest_{R_0}, Janitor_{R_0}\}$. For example, applications that allow access to $Student_{R_0}$ will not grant $Employee_{R_1}$ access unless an association $Employee_{R_1} \mapsto Student_{R_0}$ is provided. This is a demonstration of a partial mapping.

This model is highly flexible because it allows the security officer to specify the placement of specific foreign roles in the hierarchy, without enforcing a mapping for each and every role in the foreign hierarchy.

2.1.2 Dynamic translation

If new roles are added to the local or foreign hierarchies, these roles automatically fit into the translation model and the security officer does not need to make any immediate changes. Hence the policy framework is dynamic in nature. By following an appropriate

notification protocol for the deletion and addition of roles (see Section 2.2.2), the role translation model will dynamically respond to such changes.

2.2 Security issues

Once we have a role translation policy defined, one can ask the question, "How secure is this system?" We must first spell out the underlying assumptions for such a policy. This policy framework does not specify how a secure context is established. Our role translation model provides a meaningful translation of foreign roles into local roles. This is a key step that must be followed before establishing a secure context across domains. Hence, our model is useful in establishing *meaningful* secure contexts. Once our model provides the translation, it is the responsibility of the overlying security mechanisms to use the translation meaningfully.

The question of how secure this system is can only be answered in context of the global security policy that uses our role translation policy as an enhancement. For example, our model allows the security officer to give higher access to certain foreign principals than others. This could potentially make the system less secure if the foreign domain is not trusted. However, one can view our model as a way of giving lower access to certain foreign principals than others and hence the security of the system is enhanced.

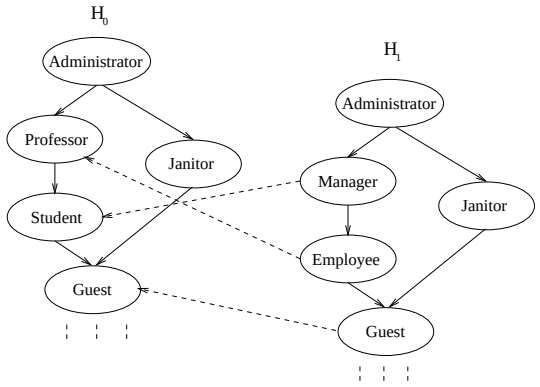

Figure 2.2 Conflicting associations

2.2.1 Conflict resolution

It is easy to imagine a situation in which a particular association conflicts with another association. For example, consider Figure 2.2. If we had a partial policy with the association $Manager_{R_1} \mapsto Student_{R_0}$, it is clear that $Employee_{R_1}$ will be translated to $Guest_{R_0}$ and not $Student_{R_0}$. Now consider a second association $Employee_{R_1} \mapsto Professor_{R_0}$. Now, $Employee_{R_1}$ will be translated to $Professor_{R_0}$. We can see that these two associations conflict since now $Manager_{R_1} > Professor_{R_0}$, even though the association $Manager_{R_1} \mapsto Student_{R_0}$ does not allow this. In such situations, conflicts are resolved by giving the foreign role the highest possible translation in the local hierarchy that the associations can allow. In other words, we take a *union* of all the reachable roles and hence do not violate the semantics of a partial ordering. Hence $Manager_{R_1}$ will be translated into $Professor_{R_0}$ since $Professor_{R_0} > Student_{R_0}$.

This may result in a security hazard in which the security officer may grant a foreign role higher access without meaning to do so. To prevent such a situation, our Role Editor

can be put into the *reachability mode*. In this mode, the security officer can select foreign roles and a reachability graph will be drawn for that selection. More specifically, the Role Editor color codes all the local roles that are *reachable* from the selected foreign role.

2.2.2 Deletion of roles

When a role is removed from the local hierarchy, we must maintain the *ancestral invariance* of the hierarchies. We define *ancestral invariance* for our role translation model as follows:

Definition 2.2.1 *Consider the association* $x_{R_1} \mapsto a_{R_0}$. *Hence,* x_{R_1} *is an ancestor of all the children of* a_{R_0}. *This ancestral relationship must be maintained even after* a_{R_0} *has been deleted. Similarly, if* $y_{R_1} > x_{R_1}$, *then* y_{R_1} *is also an ancestor of all the children of* a_{R_0}. *Again, this ancestral relationship must be maintained if* x_{R_1} *is deleted. We call this property* ancestral invariance.

To maintain the ancestral invariance, when a role is deleted in the *local hierarchy*, for every association that connects to the removed role, a new set of associations originating from the same forign role, is added to all the children of the removed role. This makes sure that the ancestral invariance is maintained even after the role has been deleted. Transitive associations are replaced by a set of transitive associations, and non-transitive associations are replaced by a set of non-transitive associations. For example, in Figure 2.1, if $Professor_{R_0}$ is removed from H_0, then the association $Manager_{R_1} \mapsto Professor_{R_0}$ is replaced by $\{Manager_{R_1} \mapsto Student_{R_0}\}$.

When a role is removed from the *foreign hierarchy*, for every transitive association that begins at the removed role, a set of new transitive associations are added from all its parents to the role that it was connected to. For example, in Figure 2.1, if $Manager_{R_1}$ is removed from H_1, the association $Manager_{R_1} \mapsto Professor_{R_0}$ is replaced by $\{Administrator_{R_1} \mapsto Professor_{R_0}\}$. Non-transitive associations are ignored. Note that for this to be feasible, if there is a change in the role hierarchy of a particular domain, all the domains that it interoperates with must be notified through some protocol.

2.2.3 Domain crossing

When a principal attempts to interoperate with a target in another domain, it must *cross* one domain boundary. We call this a *domain crossing*. As outlined below, multiple domain crossings can be a security hazard because it may allow *infiltration* and *covert promotion*. The reason why these problems arise is because we argue that the translation of roles is a non-transitive property over multiple domains. For example, a translation from domain D_2 to D_1 is not meaningful to D_0, even if there is a translation policy between D_1 and D_0. This is similar to the notion of *trust*. If A trusts B, and if B trusts C, then this does not imply that A trusts C. Similarly if $x_{R_2} \mapsto y_{R_1}$ and $y_{R_1} \mapsto z_{R_0}$, then this does not imply that $x_{R_2} \mapsto z_{R_0}$. We call this property the *inter-domain non-transitivity of role translations*.

Infiltration: Consider the case when domains D_2 and D_1 decide to interoperate, and domains D_1 and D_0 decide to interoperate. This does not imply that D_2 and D_0 want to interoperate. Even though D_0 may not want to interoperate with D_2, principals from

D_2 could first enter D_1 and consequently infiltrate into D_0. Since our model assumes inter-domain non-transitivity for role translations, our role translation model will not allow infiltration.

Covert promotion: Another problem with such domain crossings is that principals can cross domain boundaries and possibly return to the original domain with a role higher than their original role. Effectively, a principal can *covertly promote* itself in the role hierarchy by crossing multiple domains. Since our model assumes inter-domain non-transitivity for role translations, our role translation model will not allow covert promotions.

To prevent infiltration and covert promotion, the translation is made valid for only one domain crossing. Each domain translates the principal's *original* role, and not simply the role from the previous domain. This ensures that irrespective of domain crossings, the role translation model will be valid. Hence, infiltration and covert promotion can be prevented by including the original domain and role names within the principal's certificate. However, this requires the cooperation of all the domains because rogue domains could fabricate a principal's certificate to appear to have originated from that domain. Without cooperation, it is possible to have covert promotion, and infiltration.

CHAPTER 3

THE ROLE EDITOR

The Role Editor (Figure 3.1) is the security officer's tool to manage the local hierarchy, view foreign hierarchies and, most importantly, to make associations between the foreign and local hierarchies. This chapter first describes the functionality of the Role Editor, and then provides implementation details in Section 3.2.

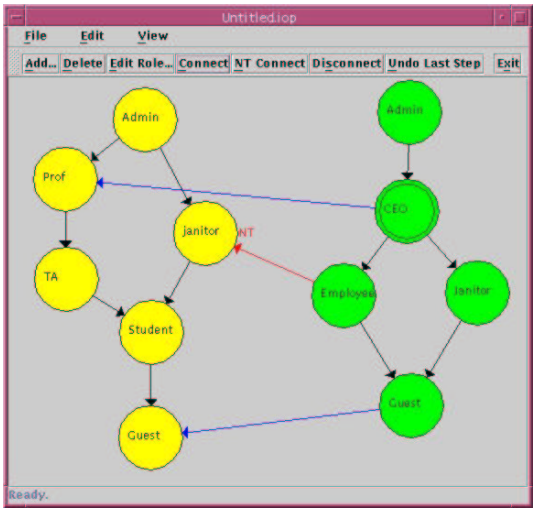

Figure 3.1 Role Editor

3.1 Functionality

Understanding the functionality of the Role Editor provides us with a basis to understand the architecture of our policy framework. The Role Editor has a plethora of features that aids the security officer in manipulating role hierarchies. These features are explained below.

3.1.1 Creating or opening a hierarchy

Role hierarchies are stored in files with the extension *.rol*. The Role Editor presents these stored hierarchies to the security officer in a graphical format. The security officer can either retrieve an existing role hierarchy by using the *Open...* command from the *File* menu, or create a new hierarchy by selecting *New...* from the same menu. When a new hierarchy is created the default root node is the *Admin* node. A hierarchy *must* have a root node. The S.O. is also prompted for the *realm-name*, which is analogous to a "domain-name."

After altering the role hierarchy, the security officer can save the hierarchy by selecting *Save* or *Save As...* from the *File* menu.

Figure 3.2 shows the local hierarchy loaded into the Role Editor.

3.1.2 Developing the hierarchy

Adding new roles: The S.O. (Security Officer) can add new roles by first selecting the parent, and then clicking on *Add...*. This will add a new role as a child of the selected

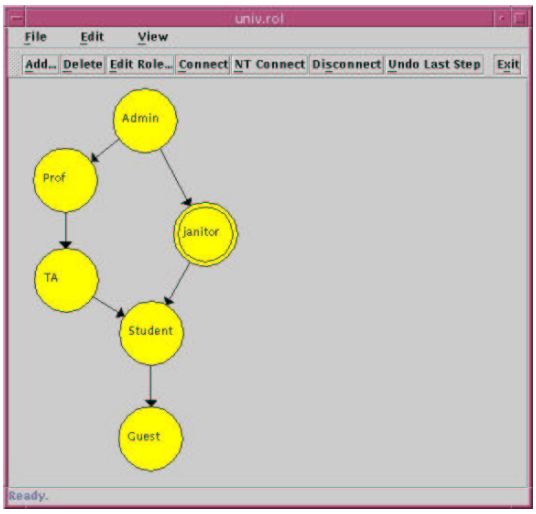

Figure 3.2 Role Editor - local hierarchy

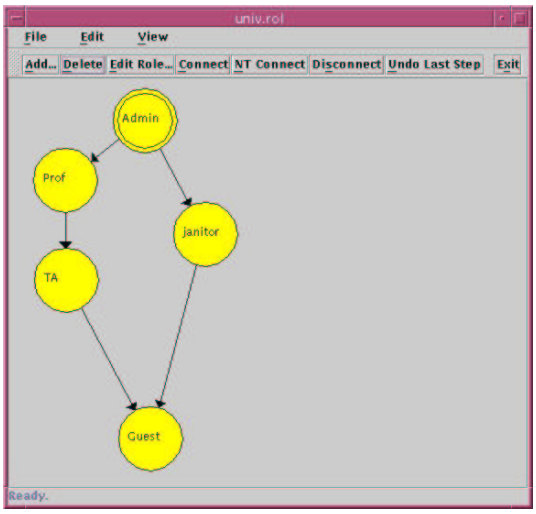

Figure 3.3 Role Editor - local hierarchy with *Student* role deleted

node. The security officer is prompted to specify a unique role name, and a risk-value[1] for this role.

Editing roles: The S.O. can select a node and then click *Edit Role...* to change the attributes (role-name and risk-value) of the selected role. The Role Editor asserts that a duplicate role-name has not been specified.

Deleting roles: The S.O. can select a node and then click *Delete* to delete a node. However, to maintain the node invariance in the hierarchy, we must connect all the parents of the deleted node, to the children of that node. This ensures that even though a node has been deleted, the parent-child relationships between other nodes are not affected. Figure 3.3 illustrates what happens when the S.O. deletes the *Student* node shown in Figure 3.2.

Connections: A parent-child relationship is asserted by adding a connection between two nodes. For example, in Figure 3.2 we can see the connection $Admin \mapsto Janitor$. A connection between two nodes can be made by first selecting a node from which the connection will originate. The S.O. then clicks the *Connect* button which indicates that a connection will be made. The S.O. then clicks on a node where the connection will end. If this new connection does not add a cycle[2] in the resultant graph, the connect operation will be accepted.

Similarly, a connection can be removed by using the *Disconnect* button. If a disconnection results in an orphaned node (i.e., if the only parent is disconnected), the root

[1]At this point, we only provide a facility to specify and store risk values. This will be used in future work for more sophisticated security models

[2]For example, a cycle in the graph can allow a role to achieve a higher status than its parent.

node is automatically made the parent of this node. Hence, it is not possible to have a node that is not reachable from the root node.

3.1.3 Specifying Associations

Loading a foreign hierarchy: Once the local hierarchy has been loaded into the Role Editor, as in Figure 3.2, the S.O. can insert the foreign hierarchy by selecting $File \mapsto Import \mapsto InsertRemote \ldots$. This brings up the foreign hierarchy next to the local hierarchy.

Adding and removing associations: Associations can be added in the same way connections are added within the local hierarchy. In addition, a *non-transitive* connection can be added by clicking on the *NT Connect* button.

3.1.4 Other Features

Undo Last Step: The S.O. can undo the last operation by clicking on this node. The Role Editor keeps track of the previous hierarchy, and simply replaces the current hierarchy with the previous hierarchy. Hence subsequent clicks on this button will toggle between the previous and current hierarchies.

Auto Arrange: In this mode, the S.O. does not have to worry about the placement of nodes in the graph. The Role Editor uses an auto-arrange algorithm to draw the hierarchy on to the Role Editor.

Manual Arrange: In this mode, the S.O. can drag nodes and hence change their locations. The Role Editor will not allow dragged nodes to overlap with other nodes and

will cancel the drag operation if it detects an overlap. When adding a new node, it will place the new node at a suitably close location, such that the new node does not overlap with another node. The S.O. can then drag this node and place it at another location.

When a hierarchy is saved the coordinates of the nodes are also stored in the file, so that the S.O.'s placement preferences are not lost.

3.2 Implementation

The Role Editor has been implemented in Java. The class diagram is shown in Figure 3.4. The individual classes are described below.

3.2.1 RoleAppFrame

This class is the outer frame of the Role Editor. It lays out the menus, and the toolbar that can be seen at the top in Figure 3.2. It also contains the functionality of enabling and disabling the Role Editor functions depending on whether a hierarchy is displayed or not. This class also interacts with the RoleGraphPanel class by keeping track of the file name. The RoleGraphPanel notifies this class of a file name change, which in turn updates the file name displayed in the frame. Observe the file name, *Univ.rol*, displayed in Figure 3.2. The RoleAppFrame carries one reference to the underlying RoleGraphPanel.

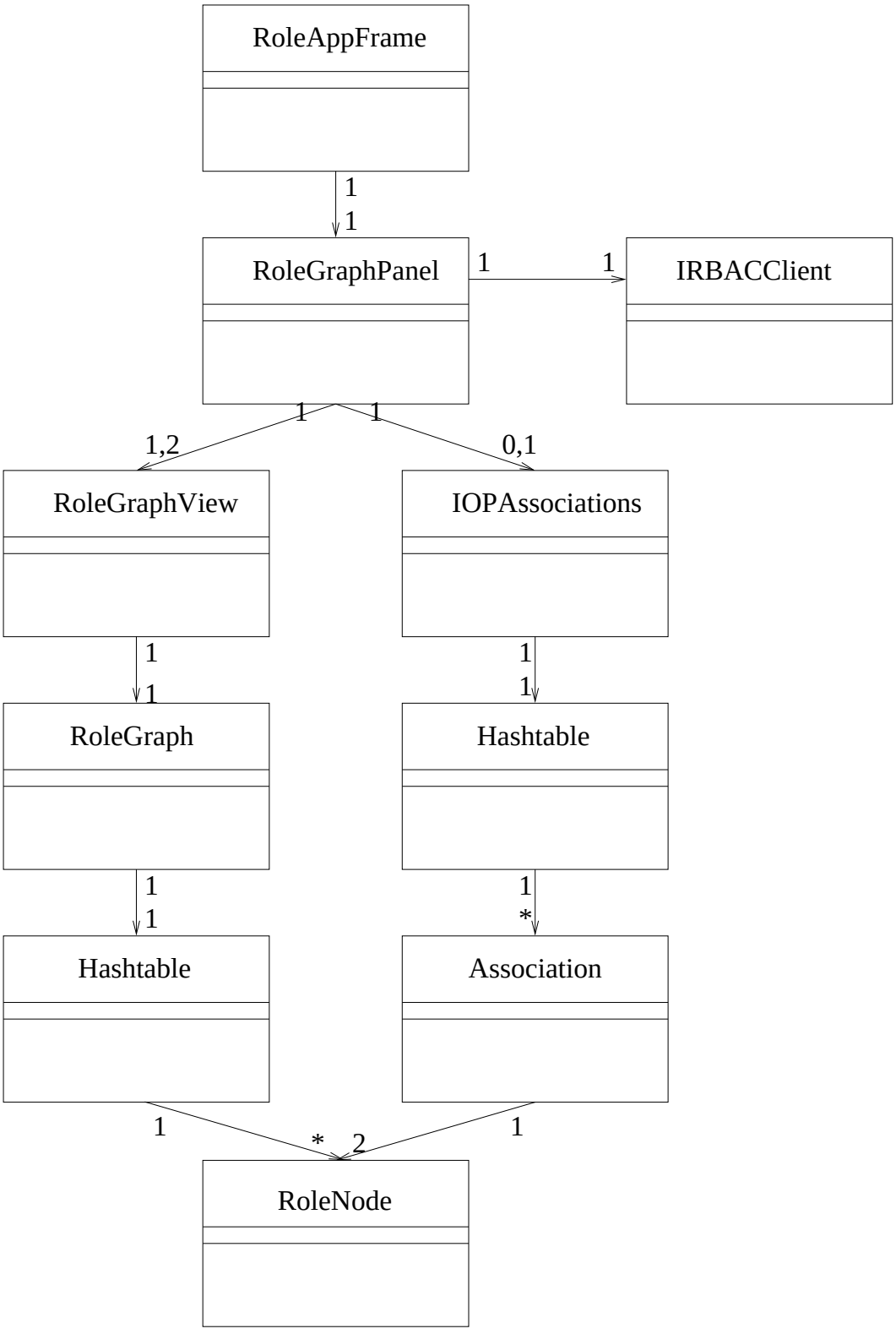

Figure 3.4 Role Editor Class Diagram

3.2.2 RoleGraphPanel

The RoleGraphPanel encapsulates most of the functionality of the Role Editor. It is responsible for opening and closing role hierarchy files (*.rol* files), and displaying the role hierarchies. It holds one, or two, references to RoleGraphView classes depending on whether it is in *interoperability mode* or not. When the Role Editor is in the interoperability mode, it displays two RoleGraphsViews and handles the associations between the two underlying RoleGraphs. When it is not in this mode, the SO can manipulate the hierarchy of a single RoleGraph. When the Role Editor is in the interoperability mode, it holds a reference to the IOPAssociations class. When not in this mode, this reference is set to null. The RoleGraphPanel also holds a reference to the IRBACClient class (Section 4.1.2), to communicate with the IRBACServer (Section 4.1.1).

The important methods in this class are described here.

- *setAutoArrange(boolean auto)*: This sets the *auto arrange mode* of the Role Editor which is described in Section 3.1.4

- *setFile(File newFile)*: This updates the new file name, and informs all the classes that have registered for the file update event.

- *paintComponent(Graphics g)*: This method is called whenever the RoleGraphPanel is painted. This method systematically draws the RoleGraphViews and the Associations depending on the interoperability mode.

- *saveFile()*: This method saves the current underlying RoleGraph to the current file name.

- *mouseClicked(MouseEvent ev)*: This method captures mouse click events. It first figures out whether a node was clicked or not. It does this by querying the underlying RoleGraphView classes with the mouse-click coordinates. If a node was clicked, then there are several cases that are handled, such as, deleting a node, connecting two nodes, disconnecting two nodes, and adding an association between two nodes.

- *mouseDragged(MouseEvent ev)/mouseReleased(MouseEvent ev)*: When the Role Editor is in the *manual arrange mode*, the SO can drag nodes to new locations within the RoleGraphPanel. This functionality is handled by these two methods.

- *actionPerformed(ActionEvent e)*: This method performs most of the functionality of the RoleGraphPanel because it captures various events generated by the user and acts on them. The following functionality is present within this method: opening files, saving files, adding and deleting nodes, connecting and disconnecting nodes, editing nodes, committing changes to the IRBACServer, inserting foreign hierarchies into the RoleGraphPanel, setting the auto/manual arrange mode and setting the reachability mode,

3.2.3 RoleGraphView

A RoleGraph can be visualized by using the RoleGraphView class. This class holds a reference to the underlying RoleGraph, and uses it to draw the components of the RoleGraph. It also adds the undo feature to the RoleGraph. Some of the important methods are listed below.

- *getDefaultNode()*: This returns the default node of the underlying RoleGraph. This is important because the RoleGraphPanel will require this information to add the default association between the default nodes of the foreign and local hierarchies.

- *getClickedNode(int x, int y)*: This returns the node in the underlying RoleGraph that contains the point (x, y).

- *drawReachability(Graphics g, Vector nodes)*: If a node is selected in the foreign hierarchy, the RoleGraphPanel will figure out the vector of "EntryPoints" and supply this for drawing the reachability graph.

- *draw(Graphics g)*: Draws the underlying RoleGraph by arranging the nodes according to whether it is in auto arrange mode or not, and then drawing each node.

3.2.4 RoleGraph

This class is the datastructure of a hierarchy of RoleNodes. It maintains connections between various nodes to make the hierarchy. Some of the important methods in this class are:

- *setRootNode(RoleNode rootNode)*: The RoleGraph must have a designated *root node*, which is at the top of the hierarchy. The root node is set through this method.

- *isMyNode(RoleNode node)*: The RoleGraph can be queried to check whether the supplied node is a part of the RoleGraph or not.

- *getEntryPoints(RoleNode node, IOPAssociations iopAssoc)*: Finds the entry points for 'node' into another RoleGraph by looking at the supplied IOPAssociations. See the explanation for the same function in Section 4.1.1 for more details.

- *saveToFile(File file)*: Stores the entire RoleGraph into a file using a simple text format. Hence this is easy for debugging purposes. This RoleGraph can be retrieved from the file by using the appropriate constructor.

- *addNode()*: There are various definitions for this method, but they all result in adding a node to the RoleGraph.

- *deleteNode(RoleNode node)*: Deletes the specified RoleNode from the RoleGraph.

- *calcReachability(Vector nodes)*: Calculate all the nodes reachable from the supplied Vector of nodes. These nodes are marked as reachable and hence colored pink.

- *isAncestor(RoleNode potentialAncestor, RoleNode node)*: checks to see whether *potentialAncestor* is an ancestor of *node* or not.

- *connect(RoleNode parent, RoleNode child)/disconnect(RoleNode parent, RoleNode child)*: Connects or disconnects two nodes in the RoleGraph.

3.2.5 RoleNode

This class holds the information for a particular role, i.e., the name and the risk value[3] associated with the node. Some of the important methods are:

- *isChildOf(RoleNode parent)/isParentOf(RoleNode child)*: Checks to see whether the current RoleNode is a child/parent of the supplied node.
- *addParent(RoleNode parent)/addChild(RoleNode child)*: Adds a parent/child to the RoleNode.
- *removeParent(RoleNode parent)/removeChild(RoleNode child)*: Removes the supplied node from the parent/child list of RoleNodes.

3.2.6 IOPAssociations

This class manages the associations between two files, and is responsible for reading/writing from/to the .iop files

Some important methods of this class are:

- *regenerate(RoleNode child, String oldName)*: If a RoleNode in the local hierarchy changes, then we need to regenerate any associations with this node, since the name of the node is used to index the association. Hence, we use the old node's name to retrieve the association, and then rename the association with the new name.

[3]Although we do not use the risk values associated with a role in our work, we provide it for future work involving risk assessment.

- *removeAssociationsTo(RoleNode node, Vector children)*: If a RoleNode in the local hierarchy is deleted, then we need to remove all associations to this node, and replace them with associations to the children of the deleted node.

- *addAssociation(RoleNode parent, RoleNode child)*: This method adds an association between a node in the foreign hierarchy (*parent*) to a node in the local hierarchy (*child*).

- *addNTAssociation(RoleNode parent, RoleNode child)*: This method adds a non-transitive association between a node in the foreign hierarchy (*parent*) to a node in the local hierarchy (*child*).

3.2.7 Association

This class maintains an association between two nodes. The parent is a RoleNode in the foreign hierarchy, and the child is a RoleNode in the local hierarchy.

CHAPTER 4

IRBAC Server

The IRBAC Server is responsible for maintaining the role translations specified by the Role Editor. The Role Editor communicates changes to the IRBAC Server. This server can be queried for specific role translations, and can be deployed into a wide range of access control systems with relative ease.

4.1 Architecture

The server architecture comprises of the IRBAC Server, the IRBAC Client and the abstraction of a Realm.

4.1.1 IRBACServer

The IRBAC Server uses Java RMI for its communication. The IRBAC Server defines a simple interface with the following methods:

- *addRealm(File realmFile)*: The Role Editor can inform the IRBAC Server to add a new *realm* for secure interoperability. For example, after the SO has specified the role translations for a new foreign domain, the SO selects the *Commit to server...* option in the Role Editor, which eventually calls this method on the server. The server then loads the interoperability file for this realm and is now dynamically configured to translate roles from the specified foreign domain (realm).

- *getEntryPoints(String role)*: The server can be queried for the translations of any role. Roles are formatted as *role@domain*, for example, Student@uiuc.edu. The server then looks up the realm indicated by the domain in the role string, and then queries it for the entry points into the local hierarchy. The entry points for a particular foreign role, F, is the set of *entry roles* reachable by F. *Entry Roles* are roles in the local hierarchy that have incident transitive or non-transitive associations.

- *reloadLocal()*: The IRBAC Server maintains only one datastructure for the local hierarchy. All foreign associations with foreign hierarchies are maintained through a single local hierarchy. When the local hierarchy is modified, the server must regenerate all the associations with the foreign hierarchies. Hence, the Role Editor informs the IRBAC Server of any changes to the local hierarchy by invoking this method on the server

- *isAncestorOf(String potentialAncestor, String role)*: This method checks whether *potentialAncestor* is the ancestor of *role* within a certain hierarchy. This provides

the simple functionality for RBAC, so that applications can make access control decisions based on the role hierarchy.

4.1.2 IRBACClient

This is a utility class that can be used by any class that needs to communicate with the IRBAC Server. It handles the complications of connection setup with the RMI Server. Hence it provides the programmer with simplicity, and is hence easily deployable in access control systems.

The IRBAC Client essentially presents the same interface as the IRBAC Server.

4.1.3 Realm

The Realm class is an abstraction to make things simpler for the IRBACServer. The Realm object is aware of the associations and translations for a particular realm. The IRBAC Server maintains a hashtable of all the realms. When it needs to perform a role translation for a particular realm, it looks up the realm in the hashtable and then queries the realm for the role translation.

The Realm class contains the following methods:

- *generate(RoleGraph localRoleGraph)*: This method generates the associations between the foreign hierarchy (stored within the Realm class) and the supplied RoleGraph for the local hierarchy. This method is all called while regenerating the associations when the local hierarchy has changed.

This class also defines *getEntryPoints()* and *isAncestorOf()*. The functionality is the same as described in Section 4.1.1, and is called by the IRBAC Server.

CHAPTER 5

Interoperability in Active Networks

In this chapter we will introduce the concept of *active networks* and the motivation for security. We will also discuss the need for an interoperability architecture and discuss our implementation for the same. We refer the reader to [6] for a detailed discussion on *Seraphim*, a dynamic security architecture for active networks.

5.1 Active networks

Active networks provides an infrastructure to applications for customizing their communications. Applications can inject *capsules* into the network. These capsules are like persistent objects in that they carry executable code and maintain state. Active routers install and execute these capsules on the application data dynamically. This makes it possible to deploy new protocols very quickly. Various other applications are also made possible. For example, one can have *shopping agents* that are active pieces of code that traverse the network and find the best available deal for a certain product.

However, securing this infrastructure against threats and exposures remains a major challenge. Allowing executable code at the routers opens the possibility for viruses, trojans and resource-hogs to name a few. Most of the research in this area has focussed on

- preventing malicious behavior of arbitrary user code, and
- protecting the user code and data from malicious routers

5.2 Seraphim - Securing active networks

Seraphim is a dynamic security architecture for active networks. It was developed at the University of Illinois by the Active Networking Group. We have made modifications to this architecture to include secure interoperability. In addition to authentication and encryption, *Seraphim* has provisions for Mandatory Access Control (MAC), Discretionary Access Control (DAC), Double Discretionary Access Control (DDAC) and Role Based Access Control (RBAC). Since our interoperability framework is based on RBAC, it works only under the RBAC mode. In other modes, the username is designated as *foreigner* and suitable access control decisions can be made in this case.

The *Seraphim* security architecture interacts with three distinct layers of an active router's software: the application, the Execution Environment (EE), and the NodeOS. The NodeOS is similar to the kernel of an operating system and performs resource allocation and management. The EE runs on a NodeOS and provides an interpreter, or

"sandbox" for active code and provides a means for accessing NodeOS resources. The active network installs and executes the capsule code dynamically on remote routers.

The *Seraphim* architecture can be seen in Figure 5.1.

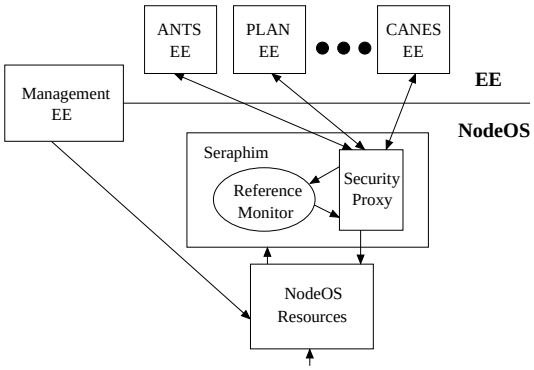

Figure 5.1 Secure Active Network Node

Seraphim is centered around the "reference monitor." The reference monitor is an extension to the NodeOS. Every node has a reference monitor through which all accesses to node resources occur. The *Seraphim* policy framework is implemented in the reference monitor, which can be downloaded dynamically when required and is hence reconfigurable.

All requests from within an EE are made through a Security Proxy, or a NodeOS Proxy, which queries the reference monitor. If the reference monitor allows the request, the call is passed down to the NodeOS; otherwise, a denial notification is sent back to the EE.

Security for a capsule is maintained by using an *active capability*. These active capabilities carry code for security policies and access control decisions. They can also verify the authenticity and integrity of the capsule's contents on a hop-to-hop basis. This also

means that the credentials can be cryptographically bound to the capsule by the active capability. Active capabilities are explained in more detail in [6].

5.3 Making Seraphim Interoperable

Various additions were made to ANTS and *Seraphim* to include interoperability. In the following subsections we describe how we added credentials to capsules and fitted our interoperability policy server into *Seraphim*.

5.3.1 Interoperable Capsules

Instead of extending the current java classes for the Capsule, we chose to create an *InteroperableCapsule* interface. This has the advantage of being compatible with any kind of Capsule definition, since an interoperable capsule only needs to *implement* this interface. If a capsule is an *InteroperableCapule* then our policy server will be used to translate roles. However, if a capsule is not an InteroperableCapsule, the normal access control decisions are made.

For a capsule to be interoperable it must implement the InteroperableCapsule interface. This interface is defined as follows:

- *public SignedIOPCredentials getOriginalCredentials();* By defining such a method, the capsule can respond to queries for its *original credentials*. These credentials correspond to the credentials of the principal in its originating domain.

- *public void setLocalCredentials(SignedLocalCredentials localCredentials);* When the capsule enters a Node in a foreign domain, the NodeOS translates the original credentials into local credentials, and stores it within the capsule using this method.

- *public SignedLocalCredentials getLocalCredentials();* When making access control decisions, the capsule can be queried for its local credentials using this method.

5.3.2 SignedCredentials

This class is the superclass for SignedIOPCredentials (original credentials) and SignedLocalCredentials (local credentials). It holds a *SignedObject* which is created using the credentials and the private key specified in the constructor. It has methods for verifying the signature of the signed object, using a supplied public key.

5.3.3 Changes to NodeOS

When a capsule enters the NodeOS, the NodeOS first checks whether the capsule is an InteroperableCapsule or not. If it is, then it retrieves the original credentials and verifies the signature of the sending domain. After the original credentials have been verified, the NodeOS then contacts the role server for the translated roles. These translated roles are then included in the local credentials, which are then signed using the local domain's private key. These local credentials are then inserted into the capsule, which will later be used in making access control decisions.

5.3.4 Changes to NodeOS Proxy

When *routeForNode()* is called on the NodeOS Proxy, the proxy checks to see whether the request has been made by an InteroperableCapsule or not. If it is, then it retrieves the local credentials, verifies the signature, and then sends the translated roles encoded in the user string to the reference monitor.

5.3.5 Changes to the Reference Monitor

Once the reference monitor receives a routeForNode() request, it decodes the user string sent from the NodeOS Proxy. If the current policy is RBAC, these roles are used in making the access control decisions. If RBAC is not in effect, then the username (*foreigner* for foreign capsules) is used.

5.4 Test application

The Ping Application was modified to use the InteroperableCapsule interface. Ping capsules that entered different nodes were translated upon entry into the NodeOS, and these translated roles were seen by the reference monitor in making its access control decisions

CHAPTER 6

RELATED AND FUTURE WORK

Very little research has been done with respect to secure interoperability between different domains. Campbell, et al. [5] discuss a security architecture for dynamic interoperability in active networks. They propose an architecture that can be used to "recursively install and support the secure deployment of new security mechanisms." In effect, they are dynamically able to install security policies on routers that may belong to different domains. This is an interesting approach where they discuss dynamic firewalls that can be used to combat denial of service attacks. Their system also operates under the RBAC model, among others, and hence our policy framework could be used to enhance their security services.

Besides active networks, our policy server was also used in enhancing the Secure InterORB Protocol (SECIOP) [2] in a Java based ORB called JacORB [4]. Secure contexts between client and target objects in different ORBs were established using our role translation mechanism [1].

Our policy server interacts with the *Role Editor* and responds to role translation requests. Future work could include a protocol to ensure inter-domain concurrency by making the policy server interact with other policy servers. Currently, there is no mechanism in place to propagate changes to the role hierarchy in one domain to other domains. Our model can also be extended to use a *risk model*. We provide a framework for specifying risk values for each role. In the future, these risk values could be used in conjunction with dynamic role translation to make better role translation decisions.

Related to our work on role translation, we have developed a prototype model of *Administrative* I-RBAC, or A-IRBAC [3]. This extends the A-RBAC [10] model of administrative hierarchies to include operations for assigning and revoking role translations.

CHAPTER 7

CONCLUSION

We have provided an efficient and dynamic method for role translation. This makes secure interoperability more flexible than conventional interoperability models that use a default or minimal translation. We believe that this model will be extremely useful in mobile networking systems, where the secure interoperability between different domains is essential.

We implemented a feature-rich application for specifying and editing role hierarchies and role translations between foreign domains and the local domain. To demonstrate our policy framework's effectiveness, we modified the current Active Network infrastructure to use our interoperability modules. We plan on releasing these changes in the next version of *Seraphim*, and hence our interoperability framework will reach a wider community.

Lastly, a conference paper outlining our I-RBAC 2000 model was published in the International Conference on Internet Computing, 2000 [7].

REFERENCES

[1] Malakim Development Pages. URL: http://choices.cs.uiuc.edu/Security/malakim/.

[2] SECIOP - Security Service Specification, December 1998. URL: http://www.omg.org/docs/ptc/98-12-03.pdf.

[3] Jalal Al-muhtadi, Apu Kapadia, Roy Campbell, and Dennis Mickunas. The A-IRBAC 2000 Model: Administrative Interoperable Role-Based Access Control. Technical Report: UIUCDCS-R-2000-2163.

[4] Gerald Brose. JacORB - a Java Object Request Broker. Technical Report B-97-02, Freie Universität Berlin, April 1997.

[5] R. Campbell, Z. Liu, D. Mickunas, P. Naldurg, and S. Yi. Seraphim: Dynamic interoperable security architecture for active networks. *IEEE OPENARCH 2000, Tel-Aviv*, March 2000.

[6] Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi. Seraphim: dynamic interoperable security architecture for active networks. In *OPENARCH 2000*, Tel-Aviv, Israel, March 26–27, 2000.

[7] Apu Kapadia, Jalal Al-muhtadi, Roy Campbell, and Dennis Mickunas. I-RBAC 2000: Secure Interoperability Using Dynamic Role Translation. In *Proceedings 1st International Conference on Internet Computing (IC'2000)*, April 2000.

[8] G. Malkin. Internet Users' Glossary - RFC 1983, network working group, August 1996.

[9] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2), February 1996.

[10] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 Model for Role-Based Administration of Roles. In *ACM Transactions on Information and System Security*, volume 2, page 105 to 135, Dec. 1999.