Napoleon
Network Application Policy Environment

D. Thomsen, R. O'Brien, C. Payne
Secure Computing Corporation
2675 Long Lake Road
Roseville, MN 55113
thomsen@securecomputing.com

Abstract

Napoleon consists of three parts; a model for specifying security policies for a heterogeneous set of network resources; a graphical tool for manipulating the model and software to translate the policy to target security mechanisms. This paper focuses on how the layered policy approach in the Napoleon model has been generalized to allow for adding additional layers. For the Napoleon tool a new approach for manipulating the role hierarchy is discussed.

1 Introduction

Napoleon is a layered approach to Role-Based Access Control (RBAC). Napoleon is named after the layered dessert, rather than the French emperor. The DARPA Information Assurance program funds the Napoleon project.

Napoleon consists of three parts

- an RBAC model for unifying diverse access control mechanisms into a single environment
- a Graphical User Interface (GUI) for manipulating that model
- software to translate the policy from Napoleon to specific access control mechanisms

This paper discusses how the original Napoleon RBAC model has evolved to incorporate semantic layers for capturing policy for different users. The paper also discusses a simplified approach for maintaining a role hierarchy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RBAC '99 10/99 Fairfax, VA, USA
© 1999 ACM 1-58113-180-1/99/0010...$5.00

1.1 Napoleon in Operation

The goal of the Napoleon effort is to provide centralized security policy management for many different access control mechanisms. Napoleon is not designed to be a centralized clearinghouse for security decisions. The applications are responsible for enforcement. Napoleon is used to load the applications with the policy they are going to enforce. There are three steps in defining a policy with Napoleon. First the Napoleon GUI is used to define the policy. Next Napoleon translates the policy to the application security mechanisms. Finally the applications are responsible for enforcing the policy, see Figure 1.

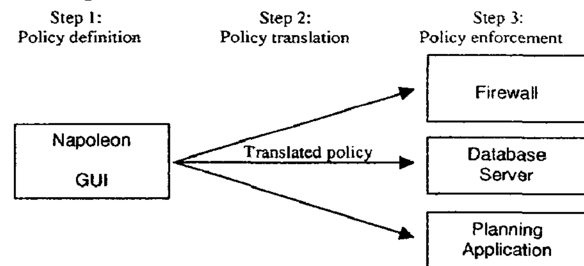

Figure 1: Steps in defining and enforcing a policy with Napoleon

The rest of this section describes the original Napoleon model for comparison purposes. Section 2 describes the new Napoleon model. Section 3 describes issues representing the Napoleon model to the user. Finally Section 4 describes the policy translation process and some of the lessons we have learned.

1.2 The Original Napoleon Model

The original Napoleon model had seven layers [1]. The model divided the task of creating the policy between two groups, the local systems administrator (local sysadmin) and the application developer. Policy creation for each group was divided into a number of sub-layers, see Figure 2.

Local System Administrator	7. Enterprise Constraints 6. Key Chains 5. Enterprise Keys
Application Developer	4. Application Keys 3. Application Constraints 2. Object handles 1. Objects

Figure 2: Seven layers of the original Napoleon model

Napoleon is based on layered sets of access. Detailed permission sets are grouped into related sets. These sets are grouped into larger sets, which may in turn be incorporated into still larger sets. Creating arbitrary sets of sets allows any policy to be expressed. However, while this offers the greatest degree of flexibility the lack of organization makes it difficult to understand and maintain the policy. To aid in understanding, Napoleon is divided into well-defined layers. Each layer has a well-defined set of semantics and constraints. More detail about these original layers will be discussed as they come up in the revised model.

As research progressed using the Napoleon model we found we were adding new layers to the model to encapsulate new semantics, such as workflow or application suites. Each new layer followed a basic pattern. Incorporating this pattern into the Napoleon model provides a generic approach to support multiple layers of policy based on semantics. Instead of dividing policy creation between two groups of users the new model allows policy creation to be split between many different groups based on semantic layers.

2 Extending Napoleon with Semantic Layers

First a brief discussion to motivate the need for adding semantic levels. The first addition to the Napoleon model was extensions to support workflow. A key component of workflow is the step. A step is the set of permissions needed to accomplish a given stage of a workflow. A step is equivalent to a mini-role. Workflow policies could be built by both the application developer and by the local sysadmin, requiring two new layers.

Experience also indicated that another layer was required to capture security policies associated with suites of applications. Applications in the suite may have common constraints and semantics. For example they may all use a clipboard to move data between applications. The pattern of accesses to the clipboard is the same for each application. The architect of the application suite is the person best suited to design the clipboard policy. The architect combines the policy components created by the application developer into a new layer that spans all the applications in the suite. This prevents the local sysadmin from having to understand the clipboard policy. Another example is a policy layer based on the environment in which the application runs. Suppose to execute in a certain environment a client application must communicate with a server. The policy interactions between the client and server are best captured in a policy layer for the system architect rather than the local sysadmin.

After adding the new layers to the original model, we were concerned that the simple Napoleon model was growing more and more complex. However careful examination of the new layers showed that the underlying structure of each layer was the same. Pieces of policy from supporting layers were combined to produce policy for higher layers. The only difference between layers was semantics. Recognizing this pattern allowed us to create a generic Napoleon model that can accommodate any number of semantic layers based on the target environment.

The new Napoleon model adopts a generic approach and instantiates the semantic layers that make sense for the target environment. For example, some enterprises may not have organized applications into suites; thus they don't need the application suite layer.

In most discussions of security policy there is an underlying assumption that a small set of users define the policy from start to finish. The Napoleon approach is that distinct sets of users maintain different parts of the policy based on their understanding and responsibilities. In the original model there were two target users, local sysadmins and application developers. The new model divides policy maintenance between any number of users. Each user combines policy pieces from the supporting layers to capture the policy constraints and semantics of their layer. These security building blocks are then available for other layers to build on.

To create a real world analogy, the building blocks are called keys. A key represents the ability to access some resource, just like in the real world where having a key allows a person to open a door. Keys become an atomic unit of the security policy. A key cannot be divided into smaller access control pieces. As shown in Figure 2 the application key forms the bottom,

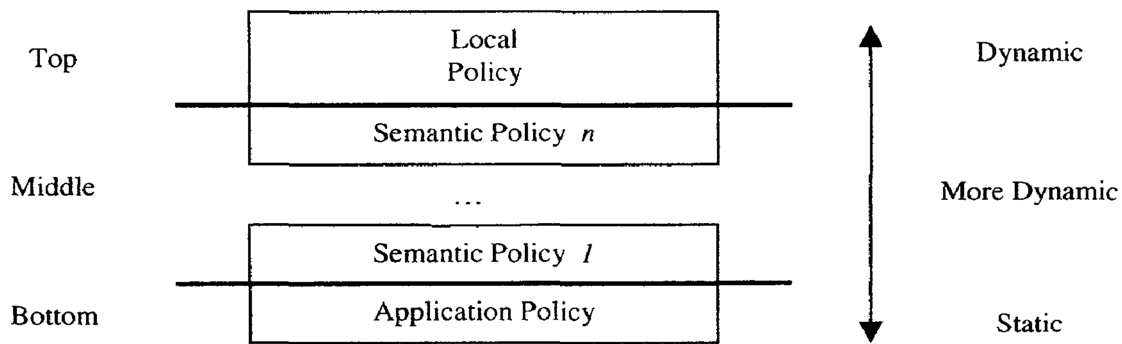

Figure 2: Generic Napoleon policy showing the general trend from static application policies to dynamic local policies

followed by any number of middle layers and finally the top layer, which binds users to the policy pieces. It is important to note that keys are not capabilities. A key is an abstract representation of some rights, independent of the implementation mechanism. A capability is data that states the bearer has the rights defined in the capability. Capabilities can be passed to other users. The Napoleon model manipulates keys to define the policy. Once the policy is defined it is translated into access control mechanisms.

Another common construct to all the layers is the concept of a key chain. A key chain is not surprisingly a collection of keys. A key chain can contain other key chains. This allows the user to create a Partially Ordered Set (POSET) equivalent to a role hierarchy. Key chains may also have constraints associated with them. If the constraint is satisfied, access in the key chain is granted, otherwise it is denied.

A final common construct to all layers is the concept of abstract key chains. The concept behind abstract key chains is very similar to the object-oriented concept of an abstract class. An abstract key chain is an intermediary grouping of keys to reflect some common policy elements. For example, there may be an abstract key chain called "health care provider" that contains permissions common to doctors and nurses. A user must never be assigned to the "health care provider" key chain rather to either a doctor or a nurse.

The generic Napoleon model much like the Napoleon dessert has a crust, any number of middle layers, and a top layer. The crust or base is the application specific access control information, the middle layers are the flexible semantic layers, and the top layer is used by the local sysadmin to assign users to the policy pieces.

2.1 The Crust (Application Layers)

The first layer of the Napoleon model is the application specific access control mechanism. The goal of this bottom layer is to encapsulate application specific information so that it can be incorporated into the higher layers in a uniform manner. This data could be Unix permission bits, Access Control Lists (ACLs) on a firewall, or sets of CORBA methods. The approach is for the application developer to use their in depth knowledge of the application to create security policy pieces that can be used to assign access to users. For example, in a health care system the application developer groups the accesses needed by a physician into a key. A doctor assigned to this key has all the necessary permission to a patient record.

Internal to the application key the policy information may be organized in any way that is convenient for the application. Ideally the Napoleon GUI would be able to display and manipulate the information in the key, but it is not required. Each key has a text description of the keys intended use, and what kind of access it grants.

To illustrate lets look at two application specific layers. To date we have created two types of application specific keys; CORBA, and CORBA enforced by DTEL++. For CORBA we developed a method for mapping Napoleon concepts to CORBASEC version 2. Unfortunately very few vendors have implemented CORBASEC version 2. For now a custom security server provides access control to methods.

Since CORBA is an emerging standard we wanted the GUI to be able to understand and manipulate the application specific access control mechanism. A CORBA application key has four sub-layers plus constraints, see Figure 3.

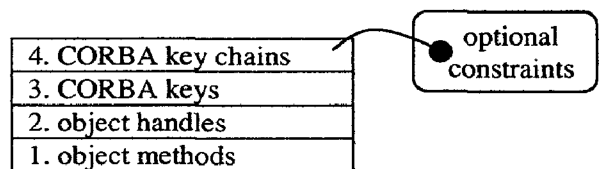

Figure 3: Application specific layers for CORBA

This is very similar to the original Napoleon model, except that constraints are no longer a separate layer [1]. Constraints are bound directly to key chains. The Napoleon GUI reads in the CORBA Interface

Definition Language (IDL) file for the application [2]. From this file the tool discovers the objects that have been defined for the application and their public methods. Methods are grouped into sets based on the semantics of the object. We call this layer handles. Handles have also been called abilities [3]. Handles in turn are grouped into keys. Keys can only contain handles from within a single IDL file to control the scope of the key. Finally key chains are groups of keys that can span several IDL files. This allows the application developer to structure their code independent of Napoleon and incorporate all the necessary privileges. Each key chain corresponds to an application role and defines the methods that are allowed to that role.

For CORBA in the DTEL++ environment the GUI is very similar. DTEL++ is NAI Labs implementation of Domain Type Enforcement for the CORBA object oriented environment [5]. In addition to controlling who can access methods DTEL++ controls who can implement the method. This is designed to protect the CORBA client from using hostile servers masquerading as legitimate servers. The key viewer for DTEL++ is identical to the standard CORBA viewer except that when a key is created it can be marked as an implement key. When the policy is translated all the users assigned an implement key get implement permission to the methods contained in the key.

As noted in Figure 3 constraints can optionally be associated with key chains. Constraints are used to capture policy information that cannot be represented as sets. Consider, for example, the fact that a role of doctor can easily describe the kinds of access a doctor needs to a patient record. However, it cannot express the fact that a doctor can only access patient records that have been assigned to them. These problems parallel the object oriented concepts of class and instance.

2.2 The Middle (Semantic Layers)

Once the application specific information has been encapsulated into a Napoleon key, it can be combined with other keys to form semantic layers. Each layer starts with a set of keys and uses them to build up key chains representing their policy. Once key chains have been built, constraints may be associated with them. The key chains for one layer become the keys of the next layer up see Figure 4. Within a layer keys are atomic units of policy. By drilling down to the next layer the user can determine how the key was composed.

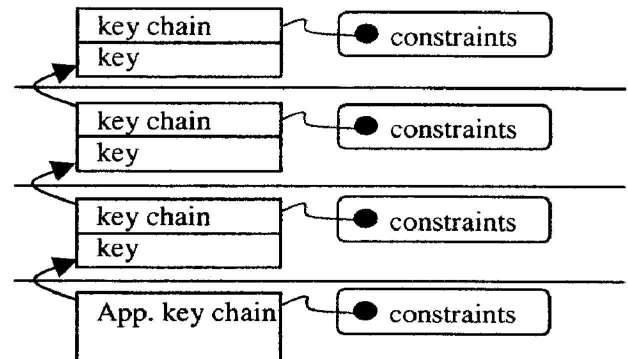

Figure 4: Interface between semantic layers

Unlike the Napoleon dessert the semantic middle layers are not just stacked one on top of the other. The relationship between semantic layers must be explicitly defined. For example, the workflow policy for a specific site may only cover the accounting and medical record applications. Thus the workflow layer only needs to use the policy components from accounting and medical records.

The Napoleon model requires each policy layer to explicitly import the policy components from the layers on which they depend. The result is much like the diagrams used to discuss layers in a software system. However, a poset more accurately describes the relationship between semantic layers, see Figure 5. As the dotted line shows the local sysadmin may need to bypass certain layers of policy to give people direct access to the firewall.

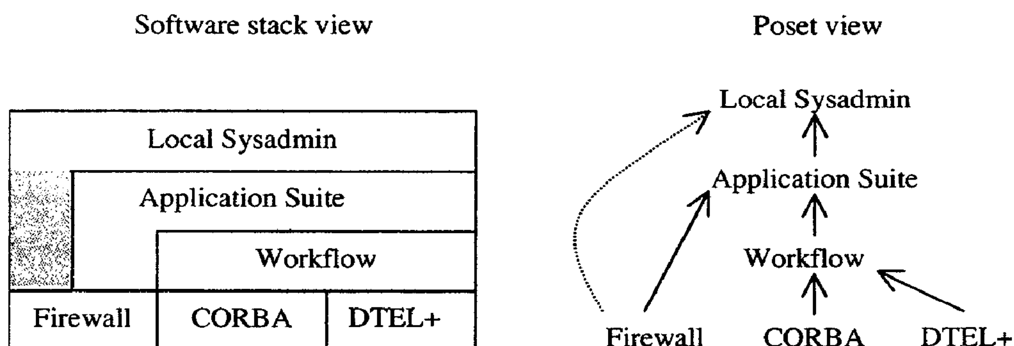

Figure 5: Two ways of looking at the relationship between semantic layers.

Since the semantic layers form a poset, a single layer could represent any policy represented in many layers. The advantage of semantic layers over a standard role hierarchy is that they impose well-defined structure.

Adding semantic layers to a role hierarchy does not increase the depth of the hierarchy. However, the depth of the hierarchy in each semantic layer is small, usually two or three. While hierarchies are excellent tools for programmers and researchers to use, a depth of seven starts to tax the limits of understanding. Deep hierarchies are even more problematic for system administrators without a programming background. Semantic layers allow users to focus on specific portions of the hierarchy increasing policy understanding.

Each semantic layer has the following properties

1. Each layer produces a set of key chains that can be exported to other layers as keys.
2. Each layer explicitly lists the other layers it is importing keys from.
3. Within a layer keys cannot be modified. Only the layer that created the key can modify it.
4. Within a layer keys can be combined to form key chains
5. Key chains can contain other key chains (from the same layer)
6. Key chains can be marked as abstract, meaning they are structural placeholders like abstract classes. In the Napoleon context what this means is that these key chains are not exported to the next layer.
7. A key chain may have constraints associated with it. If constraints are associated with a key chain the constraints must be satisfied before access is granted.

Semantic layers clearly divide responsibility for policy creation between several different users. However, it is a static type of administration. The import and export of policy components make semantic layers more static. The static nature of semantic layers has little impact because they are closely tied to static application descriptions. In fact, the application keys are part of the application interface that deals with policy. The application keys change as frequently as the application interface. Starting from the bottom of the Napoleon model there is a general trend for the lower layers to be more static because they are tied closely with the application, and the upper layers to be more dynamic.

Dynamic administration of role to role relationships in RBAC policy has been addressed in the RRA97 model [3]. In this approach administrators are given a range of roles in the hierarchy to manage. A semantic layer is equivalent to a range of roles. Many of the challenging problems in maintaining policy consistency are avoided because the new policy is installed at the same time the latest version of the application is. There is still the issue of how the changes fit into the sysadmins's policy. For example, if the sysadmin depends on a "browse" key, if the latest version of the application does not have it, the sysadmin must recreate their policy. Migration tools can be created to guide the sysadmin into choosing a new key to replace the deleted key.

2.3 The Top (Assigning Users to Roles)

The final layer of the Napoleon model is identical to the other layers except that at this level users can be associated with the key chains. The top layer is the only layer where the user role binding takes place. The top layer is also assumed to be under the control of the local sysadmin.

The top layer is more dynamic than the lower layers as it must respond to the day to day operations of the network. It is assumed that the local sysadmins are not that familiar with the applications. The local sysadmin must depend on the application developer to create policy pieces that they can use to set up their local policy. Invariably some pieces will not be sufficient. When this is the case, the user can "drill down" to the next lower layer and create a new key chain that meets their requirements.

3 The Napoleon GUI

This section looks at the issues that arise from trying to clearly display Napoleon concepts to different users with different responsibilities and varying levels of sophistication. When considering how to display policy information to a user an important distinction must be made between policies that are designed and policies that evolve over time.

3.1 Designing Policy Versus Evolving Policy

A basic premise of the Napoleon model is that the semantic levels are designed. The application developers and system architects must put as much time developing the security policy pieces as they would a good API. Application developers and system architects are familiar with object oriented hierarchies. Thus building and maintaining a good role hierarchy is a task they are well suited to do.

On the other hand the skills of the local sysadmin can vary greatly. They may have little or no experience with inheritance concepts used by the role hierarchy. More importantly sysadmins usually have a large number of responsibilities that keep them extremely

busy. As a result they do not have a great deal of time to devote to learning a new tool, and in particular they do not have time to design a role hierarchy. In fact, a role hierarchy for a local enclave can quickly change due to the introduction of new applications or policy directives. As a result, policy created by sysadmins evolves over time to meet the needs of the organization. Hence the GUI must accommodate both a design and an evolutionary approach to policy development.

The local sysadmin needs a simplified way to create and maintain the local policy. A role hierarchy is needed to express the potential policies, but a poset is a confusing data structure for the sysadmin to maintain. The most effective role hierarchies must be carefully designed, which the sysadmin does not have time to do. To simplify the GUI we propose eliminating the ability for key chains to contain other key chains for the top sysadmin layer. This results in each key chain simply having a list of keys.

This may seem like a drastic measure but let us look at it in closer detail. If the lower semantic layers have done their job, all the policy pieces should be there for the local sysadmin. As a result the role hierarchy for the top layer is very shallow. Practical experience in other environments shows that the role hierarchy is not very deep, rarely more than three [6]. For such shallow structures the benefit of the role hierarchy is small compared to the gain in simplification.

Of course simplicity comes with a cost. Lack of a role hierarchy makes three operations become more difficult

- Visualizing the relationship between roles
- Creating a new role
- Global policy changes that affect several roles

Each of these drawbacks are discussed in more detail below, as well as how a hybrid solution that creates a role hierarchy by automatically creating a partial ordering.

The drawbacks of eliminating role inheritance can be mitigated by a hybrid approach that constructs a role hierarchy from the lists of keys. Each key chain is a set of keys. The Napoleon GUI can sort the key chains into a partial ordering based on set containment. For example, a key chain with keys {a, b, c} is more powerful than a key chain with {b, c}. Key chains with the most keys appear on top, key chains with fewer keys on the bottom. Once the partial ordering is calculated the information could be shown to the sysadmin via the standard role hierarchy graph. The benefit of this approach is that the sysadmin does not have to maintain the role to role relationships explicitly, the tool constructs the role hierarchy for the user.

The first problem is visualization. A role hierarchy is an excellent way to get a quick snapshot of the relative privileges between roles. For a shallow role hierarchy visualization is probably not an issue. However the constructed role hierarchy easily can be displayed as a standard role hierarchy with all the proper visual semantics.

The second problem is creating a new role. In a role hierarchy the new role is created by first determining its parent. The role derives most of its content from the parent. Without a role hierarchy there is no parent so all of the keys for the new role have to be specifically added. To make role creation simpler without a role hierarchy we allow the user to select keys or key chains to add to new key chains. In fact we envision a general mechanism in which the user can select sets of keys and delete them or move them to a new key chain. Creating a new role starts with creating an empty key chain. The user can then select a set of keys from other key chains or a set of key chains to copy into the new key chain. Since the underlying structure is based on sets any duplicate keys are eliminated.

The third difficulty arises from the fact that low level constraints could be modified in a single place and these changes would directly impact all the senior roles. Consider the policy in Figure 6 with role inheritance. Suppose the local policy changed and all employees were allowed to browse the web. With a role hierarchy the "browse" key could be added to the employee node and the permission would automatically flow up the hierarchy. Without role inheritance there would only be three roles primary physician, consulting physician and nurse, because the abstract roles do not exist. Without role inheritance the "browse" key must be added directly to the three roles. Initially adding two extra keys does not seem like a great burden compared to eliminating the complexity of maintaining a poset.

Global policy changes could be accomplished by allowing the user to add or delete keys from the constructed role hierarchy. The tool then translates the operation from the constructed hierarchy to the underlying roles. Creating a new role could also be done using the constructed hierarchy to indicate the parent and the role's context. The constructed hierarchy obtains the advantages of the role hierarchy without the complexity of designing and maintaining it.

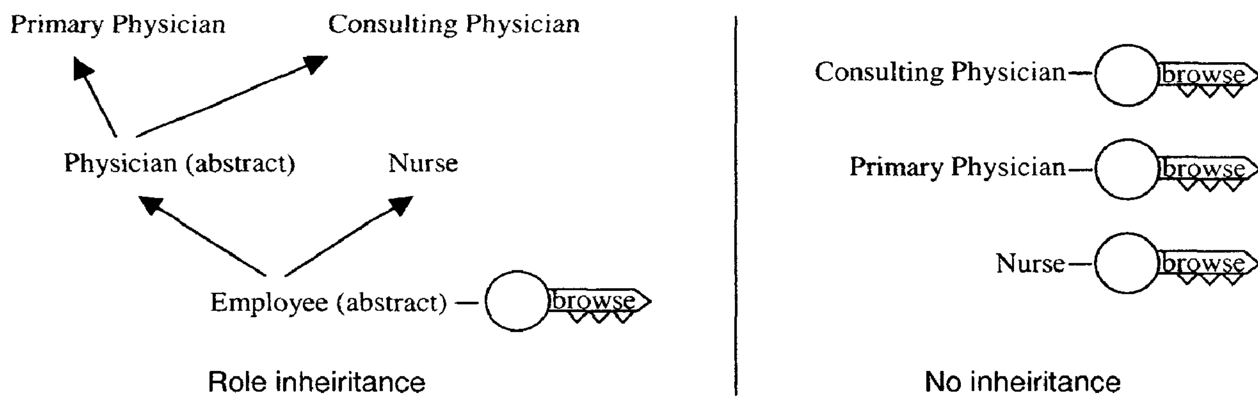

Figure 6: Comparing role inheritance

Eliminating the role hierarchy only makes sense when the policy is evolving. Clearly a designed policy is more desirable, but design takes effort and so it is best suited for a static environment so it does not have to be redesigned. A well-designed role hierarchy represents constraints, such as "all employees can access the online vending machine." However, when the GUI calculates the partial ordering there are no semantics associated with the relationship between roles.

Eliminating role inheritance simplifies maintenance only if the operations of creation of new roles, and adding global constraints are rare. If they happened frequently a role hierarchy is the best approach. Scale is another factor. Role hierarchies scale better than flat lists as the number of roles goes up. So if our assumptions about the number of sysadmin roles are wrong a role hierarchy may be a better approach. In fact, a hybrid approach is possible. A sophisticated sysadmin may create a new semantic layer just below the top layer. The new layer would have a role hierarchy for capturing the more static sysadmin's constraints. The top layer retains the simplified interface for the rapidly changing portions of the policy.

3.2 Semantic Layer Viewers

While each semantic layer has to meet the conditions outlined in section 2.2 how they are presented to the user can very greatly. The distinguishing characteristic of each layer is semantics, which implies each layer could be presented differently based on those semantics. For example, in the workflow layer the order of the steps is important to the user but not to the model. The viewer must include the step order information to provide the user with the context they need. Thus the Napoleon GUI supports each layer having its own viewer.

Potentially each semantic layer could have its own viewer. However, sometimes it is simply the grouping of keys that provides semantics, such as in the case of an application suite layer. In these cases a generic viewer is needed that provides an interface for manipulating keys and key chains. Often the cost of creating a specific viewer for a layer is prohibitive. In these cases the generic viewer can also be used.

The bottom application keys pose an interesting problem. Each security mechanism, for the most part, has already developed a way for viewing its policy. Rather than duplicate the GUI of the original mechanism in many cases it is possible to use the GUI remotely from the Napoleon GUI. For example, using the firewall GUI to manipulate user ACLs on a proxy. At other times the native viewing mechanism is too complex or does not lend itself well to being encapsulated. In such cases an opaque key can be created.

An opaque key is a construct for representing policy pieces that cannot be manipulated by the user in the Napoleon tool. The administrator cannot "drill down" into the key, only the key's description of its intended use is provided. The opaque key represents some access privilege. No access control information resides in the opaque key. The access control details are filled in when the policy is translated to the target mechanism. The opaque key approach lets the user assign predefined privileges for complex access control mechanisms.

4 Translating Napoleon Policies

Once the policy has been specified in Napoleon it must be translated to the application specific security mechanisms. The translation process works much like a compiler. A great deal depends on the security mechanism supported.

To date we have completed three translators dealing with controlling access to CORBA methods. The first translator converted the Napoleon policy to the security server created for the LOCK program. The security server was modified from its original operating system role, to provide decisions on accessing methods. The second translator targeted the

Pledge work from the Open Group [7]. The final translator targeted DTEL++ [5].

Currently the entire policy is translated to each target mechanisms. In the future we will allow the parts of the policy to translate to different mechanisms. For example, Pledge may enforce part of the policy and DTEL++ enforces the rest.

We have also designed the translator for Microsoft's COM/DCOM distributed object protocol [8]. To enforce access control on methods in DCOM, we provided DCOM interceptors that catch access requests provide fine grained access control.

Our work with policy translation has shown us two important lessons. The first is that sets provide an excellent starting point for combining and working with policy. Building a translator once the security mechanism is in place is usually a simple matter of conversion taking less that two weeks. The second lesson we learned is a relational database is useful for converting set based policies. The database allows us to construct queries to pull out the relevant pieces. For example the DTEL++ translation relies heavily on a relational database to calculate the minimum number of equivalence classes for DTEL++ types.

There are many challenges for policy translation that still must be addressed. Opaque keys appear to be a simple concept, but may cause difficulties in translation. Constraints are another challenge for policy translation. Most security mechanisms do not support any form of constraints. Constraints are critical to enforce sophisticated instance based policies. We have sketched out an approach for augmenting CORBA security with constraints, but issues remain as to how complicated the constraints can be. When the constraint is evaluated not all of the information needed by the constraint may be available.

Once the policy has been translated it is shipped out to the various access control mechanisms for enforcement. Issues remain in reconciling the central policy with the actual policy being enforced.

5 Conclusion

Research continues on enhancing the Napoleon model. The addition of semantic layers simplifies the structure and allows the model to clearly divide the process of creating security policy among several different users. One of the benefits of the Napoleon model is the encapsulation of application specific security mechanisms into a unified environment. The Napoleon GUI is experimenting with a flexible approach combining the role hierarchy with a simplified non-hierarchical layer for the systems administrators. The Napoleon tool should greatly simplify the task of policy creation and maintenance for the over worked systems administrator.

References

[1] D. Thomsen, R. O'Brien, J. Bogle. "Role Based Access Control Framework for Network Enterprises," *Proceedings of the 14th annual Computer Security Applications Conference*, pp. 50-58, December 1998.

[2] R. Orfali, D. Harkey, "Client/Server Programming with Java and CORBA," Wiley, New York, 1998.

[3] R. Sandhu, Q. Munawer. "The RRA97 Model for Role-Based Administration of Role Hierarchies," *Proceedings of the 14th annual Computer Security Applications Conference*, pp. 39- 49, December 1998.

[4] T. Keefe, "Mapping Napoleon Concepts to CORBASEC V2," internal report, Secure Computing Corporation, November 1998.

[5] D. Sterne, G. Tally, C. McDonell, P. Pasturel, D. Sames, D. Sherman, E. Sebes, "Scalable Access Control for Distributed Object Systems," to appear in Proceedings of the 8th USENIX Security Symposium, Washington, DC, August 1999.

[6] R. Awischus, "Role Based Access Control with the Security Administration Manager (SAM)," *Proceedings of the second ACM Workshop on Role-Based Access Control*, pp. 61-68, November 1997.

[7] M. Zurko, R. Simmon, "User-Centered Security," *New Security Paradigms Workshop*, September 1996.

[8] R. Sessions, "COM and DCOM: Microsoft's vision for Distributed Objects," Wiley, New York, December 1997.