

An Authorization Framework for Metacomputing Applications

T.V. Ryutov, G. Gheorghiu and B.C. Neuman
Information Sciences Institute
University of Southern California
4676 Admiralty Way suite 1001
Marina del Rey, CA 90292
{tryutov, bcn}@isi.edu
grig@imsc.usc.edu.
(310)822-1511 (voice) (310)823-6714 (fax)

February 9, 1999

Abstract

To span administrative boundaries, metacomputing systems require the integration of strong authentication and authorization methods. The problem is complicated because different components of the system may have different security policies. This paper presents a distributed model for authorization that we have integrated with the Prospero Resource Manager, a metacomputing resource allocation system developed at USC. The integration of authorization with PRM was accomplished through the specification of a policy language and the use of a Generic Authorization and Access-Control API (GAA API). The language supports the specification of diverse authorization policies including ACLs, capabilities and lattice-based access controls. The GAA API provides a uniform authorization service interface for facilitating access control decisions and requesting authorization information about a particular resource. We describe a prototype of our system.

1 Introduction

Metacomputing [8, 9] provides the abstraction of a single system for a collection of geographically dispersed computing and communication resources (e.g.

supercomputers and high-speed networks). The user of the system is presented with a consistent and familiar interface that hides the geographical scale, the complexity, and the system heterogeneity.

A metacomputing system will often cross administrative domains and involve many computing nodes. Such systems have unique requirements for security which are difficult to satisfy because of the diversity of administration and the heterogeneity of the resources involved. This difficulty stems from the variety of representations of access control policies across applications and administrative domains.

This paper describes the integration of authentication and authorization mechanisms with the Prospero Resource Manager (PRM [1]), a scalable resource allocation system that manages processing resources in metacomputing environments. PRM uses Kerberos [3] to achieve strong authentication and uses a new distributed authorization model described in this paper. Because the system must support heterogeneity in the security services supported for authentication of principals (e.g. Kerberos, DCE, SSL), we designed our method of integration to be extensible and support a variety of security services in addition to Kerberos. Further, integration of security is simplified because the acquisition of authentication and authorization credentials is handled by the transport

protocol, relieving application programmers from the need to exchange credentials within the application protocol.

Our framework consists to two components, a policy language and the Generic Authorization and Access-Control API.

- Policy language

The language allows us to represent existing access control models (e.g. ACL, capability, lattice-based access controls) in a uniform and consistent manner. Authorization restrictions allow one to define what operations are allowed, and under what conditions (e.g., user identity, group membership, time of day, or security level) particular rules apply. These restrictions may implement application specific policies.

For our integration with PRM, the restrictions include strength of authentication, limits on the physical resources managed by the system (e.g. CPU load, memory usage) and characteristics of applications that may run on a particular processor (e.g. name, version, endorser).

- Generic Authorization and Access-Control API (GAA API)

A common access control API facilitates the integration of authentication and authorization with applications. This API allows applications to request the authorization policy information for particular resource and to evaluate this policy against credentials carried in the security context for the current connections. PRM invokes GAA API functions to determine if a requested operation or set of operations was authorized or if additional checks are necessary.

Ease of use and configurability are important issues to be considered for any resource management system. For this reason, we developed a scalable mechanism based on the Prospero Directory Service to facilitate the management of the policies.

The paper is organized as follows. Section 2 describes the Prospero Resource Manager. Section 3 presents the motivation for our new authorization model applied to metacomputing applications.

Section 4 discusses the two components of the distributed authorization model: the policy language and the GAA API. Section 5 shows how the model is adapted and integrated within PRM. Section 6 describes the management of the policies using the Prospero Directory Service. Section 7 shows how the model can be applied to the lattice-based meta-computing environment. Section discusses 8 related work.

2 The Prospero Resource Manager

The design of the Prospero Resource Manager was guided by the Virtual System Model [1], in which resources of interest are readily accessible and those of less interest are hidden from view.

PRM applies this concept to the problem of allocating resources in large scale systems by dividing the functions of resource management between three types of managers: the system manager, the job manager and the node manager. Each manager makes scheduling decisions at a different level of abstraction and this separation of functions enables PRM to scale as the number of managed resources increases.

Throughout the paper, we will use the term *node* to denote a processing element, be it a processor in a multiprocessor environment or a workstation whose resources are made available for running jobs. A *job* consists of a set of communicating tasks, running on the nodes allocated to the job. A *task* consists of one or more threads of control of an application, together with the address space in which they run.

2.1 The system manager

In PRM, the total collection of processing resources is divided into subsets which correspond usually to administrative domains. Each such subset is managed by a *system manager (SM)* which is responsible to allocate its resources to jobs as needed. The system managers themselves can be organized in a hierarchical manner in order to avoid bottlenecks and ensure scalability.

The system manager maintains information about the characteristics of each resource it is responsible for, together with the mapping from resources to jobs assigned to them. The system manager receives status updates from node managers (e.g. availability, load information) and uses them to make allocation decisions. The system manager also responds to resource requests from job managers.

2.2 The job manager

The *job manager (JM)* offers a single point of contact for applications to request necessary resources. It hides from the application the complexity of managing the resources that have been allocated by the system manager to a particular job.

When a job is initiated, the job manager locates system managers (by using either the Prospero Directory Service if available or a configuration file) and sends resource requests. If the response from the system manager is affirmative, then the job manager allocates the resources to the tasks in the job and contacts the node manager for each resource in order to execute the tasks on the appropriate processors. If a system manager refuses the request, for example when the job manager is not authorized to make the request or when there are no resources available, then the job manager contacts other system managers which can satisfy its request.

2.3 The node manager

Each resource in the system is managed by a *node manager (NM)*, which is informed by the system manager about job managers that are authorized to use the resource. When the node manager receives a request from an authorized job manager, it responds by loading and executing the requested program. The node manager sends messages to the job manager about correct termination or failure of tasks and to the system manager about the availability of the node for future assignments.

2.4 Running a job with PRM

When an application that executes under PRM is invoked, a job manager is automatically started on the workstation. Figure 1 shows the flow of control: the job manager determines the resource requirements of the job and sends resource requests to one or more system managers (1); if the requested resources are available the system manager informs the node manager responsible for each resource that the resource has been assigned to particular job manager (2); the system manager returns a list of the assigned resources to the job manager (3); the job manager allocates assigned resources to the job's tasks and contacts the node manager to invoke the application (4); upon receipt of a request from the authorized job manager, each node manager loads the application task (5).

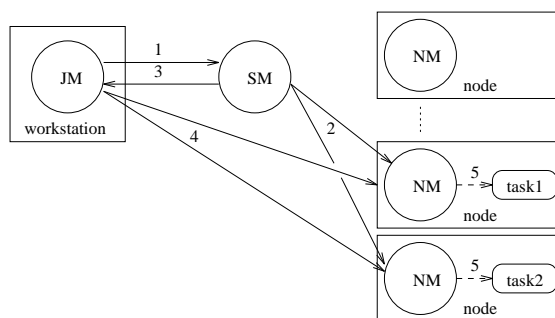


Figure 1 Running a job with PRM

3 Motivation for a New Model of Authorization

In a metacomputing environment, an application may acquire and utilize heterogeneous resources such as hosts, databases, scientific data sources, auxiliary devices (e.g. printers) and computer-controlled scientific instruments. Different access rights are defined for various resources, for instance access modes for a host may include checking the load status of the host and loading a job, while access rights for a scientific instrument may include monitoring and setting various parameters.

Our design goal is to support authorization for different kinds of resources in a uniform manner. The type of the protected resource is opaque to the authorization model, as compared with the CRISIS architecture [7], which provides ACLs where types of access are related to the type of protected resource. A file ACL lists principals allowed read, write or execute access to the file. A node ACL contains principals allowed to run jobs on this node. This approach requires code to recognize the type of protected object. Addition of new types of resources will require modifications to the existing code.

Metacomputing systems cover large networks connecting multiple research, education, government and commercial organizations. These organizations represent different administrative domains and administrators impose domain-specific security policies.

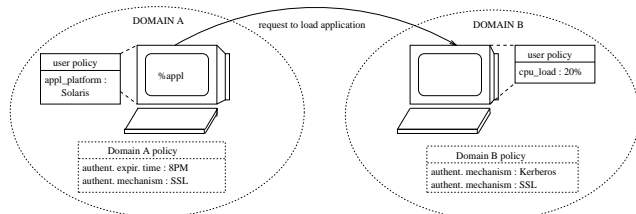


Figure 2 Multipolicy environment example

Consider the following scenario shown in Figure 2. A user logs onto a machine residing in domain A and wants to perform some computation on a remote machine residing in security domain B. Let us identify the security issues to be considered:

- 1) Establishing a trust relationship by means of authentication of the entities between security domains.
- 2) Defining fine-grained access control and authorization policies to protect client resources.

In a wide area network, it is unlikely that sites would make their resources available to others if there are no means of protection. There must be a flexible mechanism to represent user-defined security policies, such as:

- type and amount of resources that the node is willing to allocate, e.g. memory, processors, ter-

minal access, access to the local files and directories, network connections

- applications that can be run on the node, e.g. name of application, version, platform, endorser
- time periods for which access is granted, e.g. time of day or day of the week
- requirement of payment for the resources consumed

Domain administrators will define domain-specific policies as well as mandatory policies which must not be overridden by users, such as:

- authentication mechanism or set of suitable mechanisms, used to authenticate a user
- location, authorization is granted to users residing in specific hosts or domains
- attributes of users that must be possessed by users in order to get access to resource, e.g. level of competence, see Section 7

3) Specification of access policies for entities from multiple administrative domains poses additional problems:

- There are multiple mechanisms for authentication of users in different domains. Therefore, there may be no single syntax for specification of principals. For example, an application may use Kerberos V5 [3] as an authentication service. Kerberos V5 provides secret-key based authentication and format of the Kerberos V5 principal name is `user_name/instance@REALM`. Other domains may use DCE to obtain user's identity credentials. In most implementations, principals are identified by User ID and groups are identified by Group ID. Another domain might use client authentication in SSL based on public-key cryptography. Principals are identified by a global name, syntactically tied to the X.500 directory.
- Similarly there may be no standard for security policy representation. Administrators of each

domain might use domain-specific policy syntax and heterogeneous implementations of the policies.

- In a multipolicy environment the policy integration should incorporate diverse authorization models, which can coexist in distributed system, such as Access Control Lists, capabilities and lattice-based policies.

4) Discovering policies associated with the targeted resource. The assumption that all relevant credentials are passed for evaluation contradicts privacy requirements. It might be desirable to reveal only required group memberships and user attributes.

5) Access control decisions are made locally by the owner of the resource. This will require integration of different sets of policies associated with the domain providing resources, the domain requesting resources and individual users within each domain. In the example illustrated in Figure 2 the request to load an application will be granted if all depicted policies are met.

6) Enforcement of the security policies. There should be a mechanism for monitoring execution of the program on a particular node to ensure that the program keeps strictly to the limits imposed by the local administrators.

This paper describes an authorization framework designed to meet these needs. Our framework is usable for a wide range of systems and applications. It includes a flexible mechanism for security policy representation and provides the integration of local and distributed security policies. The system supports the common authorization requirements but provides the means for defining and integrating application or organization specific policies as well.

4 Overview of the Model

Our model is designed for a system that spans multiple administrative domains where each domain can impose its own security policies. It is still necessary that a common authentication mechanism be supported between two communicating systems. The

model we present enables the syntactic specification of multiple authentication policies, but it does not translate between heterogeneous authentication mechanisms. Metacomputing applications require the ability to exploit diverse resources. The policy language presented in the next section is used to specify access policies for different kinds of resources.

4.1 Policy Language

One may think about the security policy associated with a protected resource as a set of operations which a defined set of principals is allowed to perform on the target resource, and optional constraints placed on the granted operations. For example, a system administrator can define the following security policy to govern access to a printer: "Joe Smith and members of Department1 are allowed to print documents Monday through Friday, from 9:00AM to 6:00PM". This policy can be described by an ACL mechanism, where for each resource, a list of valid entities is granted a set of access rights. The same policy can be implemented using a capability mechanism. However, traditional ACL and capability abstractions should be extended to allow conditional restrictions on access rights.

Therefore, in implementing a policy, it should be possible to define:

- 1) access identity
- 2) grantor identity
- 3) a set of access rights
- 4) a set of restrictions

Policy language represents a sequence of tokens. Each token consists of:

- **Token Type**

Defines the type of the token. Tokens of the same type have the same authorization semantics.

- **Defining Authority**

It indicates the authority responsible for defining the value within the token type.

- **Value**

The value of the token. Its syntax is determined by the token type. The name space for the value is defined by the **Defining Authority** field.

The rest of this section describes the user-level representation of the policy language tokens, which can be used to implement both ACLs and capabilities. A more precise syntax is given in the Appendix.

4.1.1 Specification of Access Identity

The access identity represents an identity to be used for access control purposes. The authorization framework supports the following kinds of access identity: **USER**, **HOST**, **APPLICATION**, **GROUP** and **ANYBODY**. Where **ANYBODY** represents any entity regardless of authentication. This may be useful for setting the default policies.

The framework supports multiple existing principal naming methods. Different administrative domains might use different authentication mechanisms, each having a particular syntax for specification of principals. Therefore, **Defining Authority** for access identity indicates the underlying authentication mechanism used to provide the principal identity. Value represents the particular principal identity.

4.1.2 Specification of Grantor Identity

The grantor identity represents an identity used to specify the grantor of a capability or a delegated credential. Its structure is similar to the one of the access identity described in the previous subsection.

4.1.3 Specification of Access Rights

All operations defined on the object are grouped by type of access to the object they represent, and named using a tag. It must be possible to specify which principals or groups of principals are authorized for specific operations, as well as which principals are explicitly denied authorizations, therefore we define positive and negative access rights.

4.1.4 Specification of Restrictions

Restrictions specify the type-specific policies under which an operation can be performed on an object. A restriction is interpreted according to its type. Restrictions can be categorized as generic or specific. A restriction is generic if it is interpreted by the GAA API. For example: time of day, authentication mechanism, payment. Specific restrictions are interpreted by the application: CPU load, memory usage, applications that are to be loaded on the node.

4.1.5 Extended Access Control Lists (EACLs)

Extended Access Control Lists (EACLs) extend the conventional ACL concept by allowing specification of conditional authorization policies, implemented as restrictions on authentication and authorization credentials.

An EACL is associated with an object and lists principals allowed to access this object and the type of granted access. When an object to be protected is created, it requires the association of an EACL.

For example, the following EACL implements traditional ACL stating that anyone authenticated by Kerberos.V5 has read access to the targeted resource and any member of group 15 connecting from the USC.EDU domain has read and write access to the object.

Token Type: *access_identity_ANYBODY*
Defining Authority: *none*
Value: *none*

Token Type: *positive_access_rights*
Defining Authority: *local_manager*
Value: *FILE:read*

Token Type: *authentication_mechanism*
Defining Authority: *system_manager*
Value: *kerberos.V5*

Token Type: *access_identity_GROUP*
Defining Authority: *DCE*
Value: *15*

Token Type: *positive_access_rights*
Defining Authority: *local_manager*
Value: *FILE:read FILE:write*

Token Type: *location*
Defining Authority: *system_manager*
Value: **.USC.EDU*

The policy language we presented supports authorization models based on the closed world model, when all rights are implicitly denied. Authorizations are granted by an explicit listing of positive access rights.

The open world model, which is based on implicit granting of all rights, and listing of only negative authorizations can be represented in our model by including

Token Type: *access_identity_ANYBODY*
Defining Authority: *none*
Value: *none*

Token Type: *positive_access_rights*
Defining Authority: *local_manager*
Value: ***

as the final entry in an EACL. This will grant everybody all rights regardless of authentication. Denial of rights is then specified using negative rights in entries earlier in the EACL.

Restrictions placed on positive access rights have the goal of restricting the granted rights. The meaning of restrictions on negative (denied) access rights is unclear. We intend to investigate this issue, however, for the time being, we require that:

- 1) A single EACL entry must not specify both positive and negative rights.
- 2) If an EACL entry specifies negative rights, it must not have any restrictions.

If one allows both negative and positive authorizations in individual or group entries, inconsistencies must be resolved according to different resolution rules. The design approach we adopted allows the ordered interpretation [10] of EACLs. An ordered evaluation approach is easier to implement, it allows only partial evaluation of EACL and resolves the authorization conflicts. Evaluation of ordered EACL

starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorization is based on ordering. The authorizations that already have been examined take precedence over new authorizations.

Other interpretations are possible, but we found that for many such policies, resolution of inconsistencies was either NP-Complete or undecidable.

4.1.6 Capabilities

We present here an implementation of a capability, stating that the capability granted by the group "admin" grants read access if the capability is presented during the specified time period.

Token Type: *grantor_identity_GROUP*
Defining Authority: *kerberos.V5*
Value: *admin@USC.EDU*

Token Type: *positive_access_rights*
Defining Authority: *local_manager*
Value: *FILE:read*

Token Type: *time_window*
Defining Authority: *eastern_timezone*
Value: *8:00AM-5:00PM*

4.2 GAA API

The GAA API is used by applications to decide whether the subject is authorized for access. In this subsection we provide a brief description of the GAA API routines.

4.2.1 GAA API functions

- ***gaa_get_object_policy_info***

This function is called to obtain the security policy associated with the object. In the ACL-based systems, this information represents the object ACL, whereas in a capability-based systems, this information may contain a list of authorities allowed to grant capabilities. If no security information is attached to the object, then this function can be omitted.

- `gaa_check_authorization`

This function tells the application server whether the requested operation or set of operations is authorized, or if additional application-specific checks are required. It returns the code `YES` if all requested operations are authorized, `NO` if at least one operation is not authorized, `MAYBE` if there are some unevaluated restrictions and additional application-specific checks are required. A list of restrictions is also returned, each restriction being marked as evaluated or not evaluated.

The application must understand the restrictions that are returned unevaluated, otherwise it rejects the request. If the application understands the restrictions, it checks them against the information about the request, the target object, or other environment conditions to determine whether the restrictions have been met.

- `gaa_inquire_object_policy_info`

This function allows the application to discover access control policies associated with the targeted object applied to particular principal. It returns a list of rights that the principal is authorized for and corresponding restrictions, if any.

4.2.2 GAA API Security Context

The security context is a GAA API data structure. It stores information relevant to access control. Some of its constituents follow:

Identity Identity represents verified authentication information, such as access identity for a particular security mechanism.

Authorization Attributes This type of attributes represents verified authorization credentials, such as capabilities, group membership, group non-membership, delegated credentials. Delegation may be implemented using *restricted proxies* [2].

Evaluation and Retrieval Functions for Upcalls

These functions are called to evaluate

application-specific restrictions, to request additional credentials and verify them.

5 Applying the Distributed Authorization Model to PRM

The objects to be protected in PRM are hosts, however our model is suitable for various applications for which the objects can be files, physical devices, databases etc.

5.1 Restrictions specific to PRM

5.1.1 Application-related restrictions

Our experience with deploying PRM on a wide scale has shown that administrators are more willing to grant access to their workstations if they can restrict access to only users or organizations they trust. Administrators should also be able to specify restrictions on the specific applications allowed to use their computing resources. We have therefore introduced restrictions specific to this type of policy. Some of these restrictions follow:

a) name of application:

Token Type: *application_name*

Defining Authority: *local_manager*

Value: *matlab*

b) name of interpreter, in case the application is written in an interpreted language:

Token Type: *interpreter_name*

Defining Authority: *local_manager*

Value: *Tcl*

c) platform the application runs on:

Token Type: *application_platform*

Defining Authority: *local_manager*

Value: *Solaris*

d) version number for the application:

Token Type: *application_version*

Defining Authority: *local_manager*

Value: *1.0*

e) endorser or certifying authority for the application:

Token Type: *application_endorser*
 Defining Authority: *local_manager*
 Value: *Globus*

5.1.2 Resource-related restrictions

Authorizing a user to run a specific application on the local resources is often not enough for system administrators. What is needed is a way to impose and enforce limits on the physical resources consumed by the applications. To specify these limits, PRM uses application-specific restrictions.

a) CPU load, expressed as maximum percentage of the CPU time that an application is allowed to use:

Token Type: *cpu_load*
 Defining Authority: *local_manager*
 Value: *20%*

b) memory usage, expressed as maximum size in Kbytes that a process can occupy in main memory:

Token Type: *mem_usage*
 Defining Authority: *local_manager*
 Value: *1024*

c) machine idle time, expressed as minimum interval in minutes that the machine has to be idle before any application managed by PRM is allowed to run:

Token Type: *idle_time*
 Defining Authority: *local_manager*
 Value: *30*

5.2 Using the GAA API in PRM

5.2.1 Creation of the GAA API security context

Due to the space limitation we omit the detailed description of how principal authorization credentials are obtained and verified.

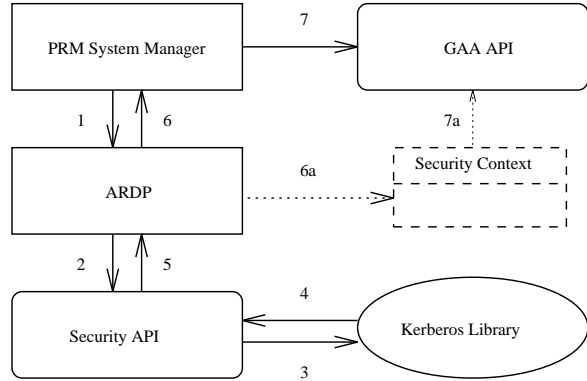


Figure 3 Creation of the security context

PRM uses calls to the Asynchronous Reliable Delivery Protocol (ARDP) [11], a communication protocol which handles a set of security services, such as authentication, integrity and payment. ARDP calls the Kerberos library through a security API, requesting principal's authentication information, which is placed into the security context and is passed to the GAA API. Figure 3 shows the flow of control: the system manager calls ARDP requesting the principal's identity (1); the request and verification of the principal's identity credentials take place (2, 3, 4, 5); ARDP places the principal's authentication credentials in the security context (6a); the system manager calls the GAA API (7); the security context, containing the verified principal's identity is passed to the GAA API (7a). When additional security attributes are required for the requested operation, the list of required attributes is returned and obtained by the application. The application or transport may add an upcall function to the security context which is passed to the GAA API and used to request additional credentials. Such additional credentials are requested, verified, and added to the security context by this upcall function.

5.2.2 Authorization Walk-through

Here we present two authorization scenarios.

First, let's consider a request from user Joe to run `matlab` on the host `kot.isi.edu` on Monday at 7:30 PM.

Assume that this host has the following ordered EACL stored in the Prospero Directory Service [4].

	IDENTITY	ACCESS RIGHTS	RESTRICTIONS		
#1	access_identity_USER	positive_access_rights	time_window	cpu_load	
DEF. AUTHORITY	KerberosV5	local_manager	psfic_tzone	local_manager	
VALUE	joe@ISLEDU	HOST:load	6AM-8PM	20%	
	IDENTITY	ACCESS RIGHTS	RESTRICTIONS		
#2	access_identity_GROUP	positive_access_rights	positive_access_rights		
DEF. AUTHORITY	KerberosV5	local_manager	local_manager		
VALUE	operator@ISLEDU	HOST:*	DEVICE:power_down		
TOKEN TYPE	access_identity_USER				
DEF. AUTHORITY	KerberosV5				
VALUE	tom@ISLEDU				
	IDENTITY	ACCESS RIGHTS	RESTRICTIONS		
#3	access_identity_ANYBODY	positive_access_rights	time_day	time_window	cpu_load
DEF. AUTHORITY	none	local_manager	local_manager	pacific_tzone	local_manager
VALUE	none	HOST:load	sat-sun	6AM-8PM	20%

Figure 4 EACL for host kot.isi.edu

When a job manager contacts a system manager with the request for resources, the system manager calls ARDP, which handles authentication as explained in Figure 3 and Section 5.2.1. If Joe is authenticated successfully, then the verified identity credential is placed into the security context, specifying Joe as the Kerberos principal `joe@ISI.EDU`.

Then the system manager calls the `gaa_get_object_policy_info` function to obtain a handle to the EACL of `kot.isi.edu`. The upcall function for retrieving the policy for the specified object from the Prospero virtual file system is passed to the GAA API and is called by `gaa_get_object_policy_info`, which returns the EACL handle.

The system manager calls the `gaa_check_authorization` function asking if `joe@ISI.EDU` is authorized `HOST:load matlab` to `kot.isi.edu`.

In evaluating the EACL, the first entry applies. It grants the requested operation, however there are two restrictions that must be evaluated. The first restriction:

Token Type: *time_window*

Defining Authority: *pacific_timezone*

Value: *6AM-8PM*

is generic and is evaluated directly by the GAA API. Since the request was issued on Monday at 7:30 PM this restriction is satisfied. The second restriction is PRM-specific:

Token Type: *cpu_load*

Defining Authority: *local_manager*

Value: *20%*

If the security context passed by PRM defined a restriction evaluation function for upcall, then this function is invoked and if this restriction is met then the final answer is **YES** (authorized). During the execution of the task the node manager is monitoring if the task is abiding to the limits imposed on the local resources and authorization time. If the corresponding upcall function was not passed to the GAA API, the answer is **MAYBE** and the set of restrictions is returned. Restrictions are marked as either evaluated or not evaluated. In our example the *time_window* restriction was evaluated and met; the *cpu_load* restriction was not evaluated and should be checked by PRM.

Next, we present an authorization scenario where additional credentials are needed. Let's consider a request from the same user Joe to shut down the node `kot.isi.edu`. The system manager calls `gaa_check_authorization` asking if `joe@ISI.EDU` is authorized to `DEVICE:power_down` host `kot.isi.edu`.

In the EACL evaluation, the first entry applies yet it does not grant the requested operation. The temporary answer is **NO** (not authorized). The second entry grants this permission, however the security context does not have group "operator" credentials. If the security context passed to the GAA API defines a credential retrieval function for upcall, then this function is invoked and if either a group membership credential for operation or delegated credential from user Tom for Joe is present, then the final answer is **YES**. If the credential retrieval upcall function was not passed to the GAA API, the answer is **NO**.

6 Managing the EACL using the Prospero Directory Service

We have mentioned in section 2 that PRM deals with scalability issues by splitting the task of managing the resources across the three types of managers. Our

goal in designing a mechanism for the management of the EACL files was to enable easy sharing of a default authorization policy among node managers, while allowing customization of the policy at the level of individual hosts.

We use the Prospero Directory Service [4] to store the information associated with the EACL files. The EACL files themselves are objects stored in the Prospero Directory Service.

The following scenario shows how the management of the files is accomplished:

1) The administrator of the domain whose resources are managed by a system manager running on host A creates an EACL file describing the default authorization policy which applies to the domain.

2) The administrator registers with the Prospero server. We supply a script which takes as input the location of the EACL file and creates a Prospero object representing a link to the EACL file, together with two attributes for the link:

```
SYSTEM_MANAGER A
EACL_DEFAULT True
```

3) If the administrator of a particular host B in the domain managed by A wants to specify a local authorization policy different from the default one, a similar procedure is followed, except that the link to the local EACL file is created with the following attributes:

```
NODE_MANAGER B
EXTEND_DEFAULT Prepend/Append/Replace
```

(**Prepend** if the local policy extends the default policy, **Append** if the local policy is appended to the default and **Replace** if the local policy completely replaces the default)

4) When a system manager is contacted by a job manager with a request for resources, it first authenticates the user, as was explained in the authorization scenario in section 5. Before requesting resources from a node manager running on a particular node B, the system manager retrieves the EACL file associated with that node by looking for a link with attribute **NODE_MANAGER = B**. If no such link is found, the default EACL file provided for the domain will be used. The file is retrieved by looking for a link with attributes **SYSTEM_MANAGER = A** and **EACL_DEFAULT = True**. If a link with **NODE_MANAGER = B** is found,

then a second query is issued for the value of the attribute **EXTEND_DEFAULT**. If the value is **Prepend** or **Append**, the system manager will have to retrieve the default EACL file first, and then prepend or append it the contents of the EACL file for node B. If the value is **Replace**, then only the EACL file for node B will be retrieved and used.

5) After retrieval of the EACL file, evaluation of the restrictions listed in the file follows, as detailed in the authorization scenario from section 5. If all the restrictions are met, the job manager is allowed to use the resources on that particular host.

6) During the execution of tasks on a particular host, the node manager periodically checks whether the task is abiding to the limits imposed on the local resources. If it is not, then the task is interrupted and the job manager is notified.

7 Lattice-based Policies

We can envision a metacomputing environment where lattice-based policies should be used to guard access to resources. Consider a metacomputing application that performs certain experiments which include interactions with scientific facilities, e.g. High-Frequency radio transmitter used for heating of the ionosphere [12]. The application tasks include performing extensive computations spread to a large number of hosts for analyzing the results and steering the transmitter (change the operating frequency, rotate the dish, turn the transmitter on or off, etc.).

The security administrator of the transmitter will establish security policy based on the level of competence to perform specific operations. To prove eligibility to access the resource, a user has to present a valid credential, stating user's competence level.

Assume there are defined three levels of competence: high, medium and low.

- 1) **High** may perform any operation
- 2) **Medium** may perform all operations except for changing the operating frequency.
- 3) **Low** is allowed only to monitor the experiments.

To implement the required lattice-based policy, a generic restriction

Token Type: *lattice_above*
Defining Authority: *security_administrator*
Value: *competence_level*

can be used. It specifies that a subject, wishing to get access to the resource has to have competence level no less than the one, specified in the **Value** field.

This is a form of lattice-based policy [13] in a sense that users are unavoidably constrained by the domain protection policy. Only a specially authorized person (e.g. security administrator of the transmitter) may change the competence level required to perform a particular operation.

There may be additional generic restrictions posed on the granted access, such as integrity message protection to ensure that the request is free from unauthorized modification, or list of trusted certifying authorities who can attest to the competence level of a requester.

8 Related Work

Nagaratnam and Byrne [16] present a model for Internet user agents (browsers, tuners, etc.) to control access to client resources. This model protects client machines from hostile downloadable content and allows the client to selectively grant access to trusted agents. The authenticity of the code is based on digital signatures of principals certifying it. All access control requests are mediated by calling a security manager component and decisions are based on the user's access control specifications stored in the policy database.

The model is restricted to using the Javakey utility as an authentication mechanism based on public key digital signatures, while our model is general enough to use a variety of security mechanisms based on public or secret key cryptosystems.

Another disadvantage of that model is the duplication of common information. Each user has to maintain a database of any principals specified in the policy database and their public keys, as well as specification of groups. These databases should be properly protected. In contrast, PRM uses Kerberos to achieve strong authentication. The authentication database is maintained centrally by the KDC

and stored on a physically secure machine. Our model also supports a group certification mechanism. A group server maintains and provides group membership information, and issues group membership and non-membership certificates. The certificates are placed into the GAA API security context and checked by the GAA API when making authorization decisions. There is no need for each user to maintain authentication and group specification databases locally.

Finally, this model can be applied only to browser-like (user agent) applications, while our model can deal with any kind of application.

The Generalized Access Control List framework described by Woo and Lam [5] presents a language-based approach for specifying authorization policies. The GACL model supports only system state-related restrictions within which rights are granted, such as: current system load, maximum number of copies of a program to be run concurrently. This may not be sufficient for distributed applications. Our model allows fine-grained control over the restrictions.

Both restricted proxies [2] and the use-condition model [15] allow conditions and privilege attributes to be embedded in authorization credentials or certificates. These mechanisms can be readily integrated with the authorization model presented here: the restrictions or conditions carried in the proxy or certificate are evaluated by the GAA API in addition to the restrictions in the matching EACL entry.

The CRISIS architecture [7] is security system based on public key cryptography. Types of access in CRISIS ACLs are related to the type of protected object. CRISIS ACLs do not support specification of constraints placed on resources that principals are allowed to consume. Access requests to an object are mediated by contacting the object's reference monitor. Reference monitors are service-specific and implemented as separate modules. The emphasis of our work is on providing a general framework for representing security policies and facilitating authorization decisions for metacomputing applications. Our model provides a uniform authorization mechanism that is capable of supporting different operations and different kinds of protected objects.

The Tivoli Management Environment (TME

10)[14] commercially available security system that uses a role-based approach to security. TME roles are named capabilities, containing a list of objects and access permissions to those objects. Objects can have default access and can be associated with more than one role. Each role will have a different level of access to the object. Roles are defined to support a particular job function within an organization, e.g. customer support or management. Groups are assigned roles, thus giving members of those groups access capabilities to the objects assigned to those roles. The TME approach can be mapped to our framework.

TME lacks flexibility in supporting user-defined security policies. It has a fixed pre-defined set of object types and generic access permissions that are available on each object type.

In addition, the TME model requires creation of a new role to include each possible combination of objects and access rights. This becomes very cumbersome for systems where a large number of operations exist on various objects.

9 Conclusions

This paper has shown that it is possible to integrate flexible distributed authorization methods and meta-computing application by extending the traditional access control lists framework with restrictions on authorized rights, and by using the Generic Authorization and Access-control API to make access decisions. The policy language interpreted by the GAA-API allows applications and users to define their own access control policy types, and supports both local and distributed security policies. The problem of translation of the policies is addressed by using generic or application-specific evaluation functions. A prototype integrating the GAA-API with the Prospero Resource Manager has been developed at the Information Sciences Institute of the University of Southern California.

10 Appendix

We use the Backus-Naur Form to denote the elements of our policy language. Square brackets, [], denote optional items and curly brackets, {}, surround items that can repeat zero or more times. A vertical line, |, separates alternatives. Items inside double quotes are the terminal symbols.

An ACL is specified according to the following format:

```
acl ::= {acl_entry}
acl_entry ::=
access_identity {access_identity}
pos_access_rights {restriction}
{pos_access_rights {restriction}} |
access_identity {access_identity}
neg_access_rights

access_identity_type ::=
access_identity_type def_authority value

access_identity_type ::=
"access_identity_HOST" |
"access_identity_USER" |
"access_identity_GROUP" |
"access_identity_APPLICATION" |
"access_identity_ANYBODY"
```

A capability is defined according to the following format:

```
capability ::=
grantor_identity pos_access_rights {restriction}
{pos_access_rights {restriction}}

grantor_identity ::=
grantor_identity_type def_authority value

grantor_identity_type ::=
"grantor_identity_HOST" |
"grantor_identity_USER" |
"grantor_identity_GROUP" |
"grantor_identity_APPLICATION" |
"grantor_identity_ANYBODY"

pos_access_rights ::=
"positive_access_rights" def_authority value
{"positive_access_rights" def_authority value}
```

```

neg_access_rights ::=
"negative_access_rights" def_authority value
{"negative_access_rights" def_authority value}

restriction ::=
restriction_type def_authority value

restriction_type ::= alphanumeric_string

def_authority ::= alphanumeric_string

value ::= alphanumeric_string

```

11 Acknowledgments

This research was supported in part by the Defense Advanced Research Project Agency under the Scalable Computing Infrastructure (SCOPE) Project, TNT, Contract No. DABT63-95-C-0095, Security Infrastructure for Large Distributed systems (SILDS) Project, Contract No. DABT63-94-C-0034, and by a grant from Xerox Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of U.S. Army Intelligence Center and Fort Huachuca Directorate of Contracting, the Defense Advanced Research Project Agency, the U.S. Government, or Xerox Corporation.

References

- [1] B.C. Neuman and S. Rao.
The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems *Concurrency: Practice and Experience*, 6(4): 339-355, June 1994.
- [2] B.C. Neuman.
Proxy-based authorization and accounting for distributed systems. *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, May 1993.
- [3] B.C. Neuman and T. Ts'o.
Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33-38, September 1994.
- [4] B.C. Neuman, S. Augart and S. Upasani.
Using Prospero to support integrated location-independent computing. *Proceedings of the Symposium on Mobile and Location-independent Computing*, pages 29-34, August 1993.
- [5] T.Y.C. Woo and S.S. Lam.
A framework for distributed authorization. *Proceedings of the ACM Conference on Computer and Communications Security, Fairfax, Virginia*, November 1993.
- [6] M. Abadi, M. Burrows, B. Lampson and G. Plotkin.
A calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, Vol. 15, No 4, pages 706-734, September 1993.
- [7] E. Belany, A. Vahdat, T. Anderson and M. Dahlin.
The CRISIS wide area security architecture. *Proceedings of the 7th USENIX Security Symposium* San Antonio, Texas, January 1998.
- [8] Edited by I. Foster and C. Kesselman.
The GRID: Blueprint for a New Computing Infrastructure *Morgan Kaufman Publishers*, 1999.
- [9] I. Foster and C. Kesselman.
Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, Summer 1997.
- [10] W. Shen and P. Dewan.
Access Control for Collaborative Environments. *Proceedings of the CSCW*, November, 1992, pages 51-58
- [11] The Asynchronous Reliable Delivery Protocol.
<http://gost.isi.edu/info/ardp>
- [12] Technical Details about the HAARP Program.
<http://w3.nrl.navy.mil/projects/haarp/tech.html>, October 1997.

- [13] S. B. Lipner.
Non-Discretionary Controls for Commercial Applications. *Proceedings IEEE Symposium on Security and Privacy*, 1982.
- [14] IBM.
TME 10 security management.
<http://www.tivoli.com/redbooks/html/sg242021/2021fm.html>, October 1997.
- [15] W. Johnson and C. Larsen.
A use-condition centered approach to authenticated global capabilities: Security architectures for large-scale distributed collaborative environments. *LBNL Report 38850*
- [16] N. Nagaratnam and S.B. Byrne.
Resource access control for internet user agent. *Proceedings of the third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon*, June 1997.