# Representation and Evaluation of Security Policies for Distributed System Services

Tatyana Ryutov and Clifford Neuman
Information Sciences Institute
University of Southern California
4676 Admiralty Way suite 1001
Marina del Rey, CA 90292
{tryutov, bcn}@isi.edu
(310)822-1511 (voice) (310)823-6714 (fax)

## Abstract

*We present a new model for authorization that integrates both local and distributed access control policies and that is extensible across applications and administrative domains. We introduce a general mechanism that is capable of implementing several security policies including role-based access control, Clark-Wilson, ACLs, capabilities, and lattice-based access controls. The Generic Authorization and Access-control API (GAA API) provides a generic framework by which applications facilitate access control decisions and request authorization information about a particular resource. We have integrated our system with the Prospero Resource Manager and Globus Security Infrastructure.*

## 1 Introduction

The conventional concept of an Access Control List (ACL) is the architectural foundation of many authorization mechanisms. A typical ACL is associated with an object to be protected and enumerates the list of authorized users and their rights to access the object. Access rights are selected from a predefined fixed set built into the authorization mechanism. Specification of the subjects is bound to the particular security mechanism employed by the system. The limitations of the traditional access control model become apparent when it is applied in a heterogeneous, administratively decentralized, distributed environment.

The variety of services available on the Internet continues to increase and new classes of applications are evolving, including metacomputing, remote printing, and video conferencing. These applications will require interactions between entities in autonomous security domains. The generic traditional access rights may not be sufficient for some applications to express authorization requirements. For example, a site might be willing to make its resources available to others, but limited to maximum CPU and memory utilization or based on a requirement for payment. It is difficult to specify such security policies in terms of conventional ACLs.

Specification of security policies for principals from multiple administrative domains poses additional problems:

- In a multipolicy environment, policy integration should incorporate the diverse authorization models that can coexist in a distributed system.
- The implementation will require integration of different sets of policies associated with the domain providing resources, the domain requesting resources and the individual users within each domain.
- There are multiple mechanisms for authentication of users in different domains. Therefore, there may be no single syntax for specification of principals.
- Administrators of each domain might use domain-specific policy syntax and heterogeneous implementations of the policies. Generalizing the way that applications define their security requirements provides the means for integration and translation of security policies across multiple authorization models.

This paper describes an authorization framework designed to meet these needs. Our framework is applicable for a wide range of systems and applications.

It includes a flexible mechanism for security policy representation and provides the integration of local and dis-

[0] In Proceedings of the DARPA Information Survivability Conference Exposition, January 2000. Hilton Head, South Carolina.

[0] 0-7695-0490-6/99 $10.00 ©1999 IEEE

tributed security policies. The system supports the common authorization requirements and provides the means for defining and integrating application or organization specific policies as well. We show how this mechanism can implement role-based access control, Clark-Wilson model, and lattice-based policies.

Our framework consists of 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 the administrator to define which operations are allowed, and under what conditions (e.g., user identity, group membership, or time of day). These restrictions may implement application-specific policies.

- Generic Authorization and Access-control API

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

## 2 Related Work

There has been recent work elsewhere on access control models for Internet user agents [7], [8]. These models apply to the Javakey utility as an authentication mechanism and use public key digital signatures. Our model is general enough to use a variety of security mechanisms based on public or secret key cryptosystems. Also, our model is application-independent whereas the models in [7] and [8] apply primarily for browser-like applications.

The Generalized Access Control List (GACL) framework described by Woo and Lam [3] presents a language-based approach for specifying authorization policies. The main goal of the GACL framework is merging policies associated with different objects and to resolve complex dependencies. GACL allows specification of the inheritance rules; access rights can be propagated from one object to the other. A gacl may reference other gacls in its entries. The benefit of the GACL approach is the ability to omit redundant information but it may require the retrieval and evaluation of more then one gacl. Specification of policy dependencies with inheritance is error-prone and may result in circular dependency of the policies and inconsistency may result.

More importantly, the expressive power of GACL is limited to that of ACL-based schemes and provides no support for capabilities and multi-level security systems. The GACL model supports only system state-related conditions within which rights are granted, such as current system load and 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 conditions.

Policy management issues were addressed by Blaze, et. al. [9] with a claim that using PolicyMaker strengthens security. Because PolicyMaker credentials bind granted rights to public keys, instead of identities, this eliminates one level of indirection. Unfortunately, this binding complicates authorization management, and as applied in cases where a system uses X.509 or PGP certificates, this binding is dependent on the application which translates credentials to the PolicyMaker format.

Policies in the PolicyMaker format are easily expressed in our framework. We treat security policies as a set of operations that subjects are allowed to perform on the targeted objects, and optional constraints are placed on the granted operations. The basic question of access control is whether a subject is allowed to perform a requested operation. The GAA API provides a common interface for asking this question. In contrast, to use PolicyMaker an application developer must define an application-specific language describing the requested operation. This language might not be reusable across different application domains.

The related work described so far presents static policy evaluation mechanisms. Decisions are based on a set of policies and credentials presented at the time of the request. In contrast, our framework allows dynamic policy evaluation where credentials can be requested from the client or from third parties during recursive evaluation of policies within the API.

## 3 Overview of the Framework

Our framework is applied to distributed systems that span multiple autonomous administrative domains without a central management authority. Applications may impose their own security policies and use different authentication services, e.g. Kerberos, DCE or X.509 certificates. We assume that within a distributed system, multiple independent applications coexist.

The individual security requirements of each application are reflected in application-specific security policies. There might exist common ACLs that apply to sets of applications. Therefore, we designed a flexible and expressive mechanism for representing and evaluating authorization policies.

It is general enough to support a variety of security mechanisms based on public or secret key cryptosystems, and it is usable by multiple applications supporting different operations and different kinds of protected objects.

The major components of the architecture are:

- Authentication mechanisms perform authentication of users and supply credentials.
- A group server maintains group membership information.
- The GAA API; Applications call GAA API routines to check authorization against an authorization model. The API routines obtain policies from local files, distributed authorization servers, and from credentials provided by the user. They combine local and distributed authorization information under a single API based on the requirements of the application and applicable policies.
- Delegation is supported by delegation credentials, such as *restricted proxies* [1], or through other delegation methods.

## 3.1 Policy Language

The security policy associated with a protected resource consists of a set of allowed operations, a set of approved principals, and optional operation constraints. 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, to do so, traditional ACL and capability abstractions must 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 conditions
The 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`

  Indicates the authority responsible for defining the value within the token type.

- `Value`

  The value of the token. Its syntax and semantics are 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. More precise syntax is given in the Appendix.

### 3.1.1 Specification of Access Identity

The access identity represents an identity to be used for access control purposes. The authorization framework supports the following types of access identity: `USER`, `HOST`, `APPLICATION`, `CA` (Certification Authority), `GROUP` and `ANYBODY`. Where `ANYBODY` represents any entity regardless of authentication. This may be useful for setting the default policies. The type of access identity is useful in determining which additional credentials are needed (see section 3.3). Principals can be aggregated into a single entry when the same set of access rights and conditions applies to all of them.

Our 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.

### 3.1.2 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.

### 3.1.3 Specification of Access Rights

It must be possible to specify which principals or groups of principals are authorized for specific operations, as well as who is explicitly denied authorizations, therefore we define positive and negative access rights.

All operations defined on the object are grouped by type of access to the object they represent, and named using a tag. For example, the following operations are defined for a file:
  `Token Type:` *pos_access_rights*
  `Defining Authority:` *local_manager*
  `Value:` *FILE:read,write,execute*

However, in a bank application, an object might be a customer account, and the following set of operations might be defined:

```
Token Type: pos_access_rights
Defining Authority: local_manager
Value: ACCOUNT:deposit,withdraw,transfer
```

### 3.1.4 Specification of Conditions

Conditions specify the type-specific policies under which an operation can be performed on an object. A condition is interpreted according to its type. Conditions can be categorized as generic or specific. Generic conditions are evaluated within the access control API; specific conditions are application-dependent and usually are evaluated by the application. These are several of the more useful generic conditions [1].

- **time**

  Time periods for which access is granted.

- **location**

  Location of the principal. Authorization is granted to the principals residing on specific hosts, domains, or networks.

- **message protection**

  Required confidentiality/integrity message protection. This condition specifies a level or mechanism that must be used for confidentiality or integrity if access is to be granted.

- **privilege constraints**

  Specifies well-formed transactions and separation of duty constraints. For more details see Section 8.

- **multi-level security constraints**

  Specifies mandatory confidentiality and integrity constraints. For more information see Section 9.

- **payment**

  Specifies a currency and an amount that must be paid prior to accessing an object.

- **quota**

  Specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained.

- **strength of authentication**

  Specifies the authentication mechanism or set of suitable mechanisms, for authentication.

- **trust constraints**

  Specifies restrictions placed on security credentials. For more information see Section 6.

- **attributes of subjects**

  Defines a set of attributes that must be possessed by subjects in order to get access to the object, e.g. security label.

If generic conditions are not sufficient for expressing application-specific security policies, applications specify their own conditions. Anything that can be expressed as an alphanumeric string can be a condition. The application must provide evaluation rules for the application-specific conditions, or be prepared to evaluate the condition once the authorization call completes.

### 3.1.5 Extended Access Control Lists (EACLs)

Extended Access Control Lists (EACLs) extend the conventional ACL concept by allowing one to specify conditional authorization policies. These are implemented as conditions on authentication and authorization credentials. An EACL is associated with an object and lists the subjects allowed to access this object and the type of granted access. For example, the following EACL implements policy 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_id_ANYBODY
Defining Authority: none
Value: none

Token Type: pos_access_rights
Defining Authority: local_manager
Value: FILE:read

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

Token Type: access_id_GROUP
Defining Authority: DCE
Value: 15

Token Type: pos_access_rights
Defining Authority: local_manager
Value: FILE:read FILE:write

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

The framework supports various strengths of user authentication. A user may be granted a different set of rights, depending on the strength of the authentication method used for identification. Specification of weaker authentication methods including network address or username will allow the GAA API to be used with existing applications that do not have support for strong authentication.

Objects that need to be protected include files, directories, network connections, hosts, and auxiliary devices, e.g. printers and faxes. Our authorization mechanism supports these different kinds of objects in a uniform manner. The same EACL structure can be used to specify access policies for different kinds of objects. Object names are drawn from the application-specific name space and are opaque to the authorization mechanism.

When a protected object is created, an EACL is associated with the object. The management of EACLs, including giving authority to modify an EACL, is supported through inclusion of entries specifying which principals are allowed to modify the EACL. The control permissions comprise a separate set of access rights named with the tag *MANAGEMENT*. To restrict the ability to pass the control permissions to others a condition *no_delegation* may be specified associated with such entries.

#### 3.1.6 Capabilities

Here we present an implementation of a capability. The example states that the capability granted by the group *admin* permits read access if the capability is presented during the specified time period.

```
Token Type: grantor_id_GROUP
Defining Authority: kerberos.V5
Value: admin@USC.EDU

Token Type: pos_access_rights
Defining Authority: local_manager
Value: FILE:read

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

### 3.2 EACL evaluation

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. Restrictions placed on positive access rights have the goal of restricting the granted rights. The meaning of conditions 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 conditions. If both negative and positive authorizations are allowed in individual or group entries, inconsistencies must be resolved according to resolution rules. The design approach we adopted allows the ordered interpretation [11] of EACLs. 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 were possible, but we found that for many such policies, resolution of inconsistencies was either NP-Complete or undecidable.

There may be interactions when independent credentials are used, e.g., one set of credentials causes denial, but the other causes accept. A user may chose to withhold credentials that it believes may result in a denial. The administrator must deal with these issues by carefully setting policies in an EACL. Conflicts may arise when more then one entry applies. For example, one matching entry specifies individual subject (user, host or application), and another matching entry specifies a certain group name. In this case, we would require the entry for the individual subject to be placed before the entry for the group (assuming the policy expressed for the individual subject entry is an exception to the policy expressed for the group entry). When several EACL entries with different conditions apply, entries for which conditions are not satisfied will not affect the outcome of the authorization function.

An ordered evaluation approach is easier to implement as it allows only partial evaluation of an EACL and resolves the authorization conflicts. The problem with this approach is that it requires total ordering among authorizations. It requires careful writing of the EACL by the security administrator and is error-prone. An improper order of the EACL entries may result in discrepancies between the intended policy and the one that results from evaluation of the EACL. It might be useful to have a separate module [4], [9], that would help verify and debug the EACL to assure that it expresses the desired policy.

### 3.3 Credential evaluation

Credentials are translated to the GAA API internal format and placed into the GAA API security context. When evaluating an EACL, the security context is searched for the necessary credentials. Assume that file *doc.txt* has the following EACL shown in **Table 1.** stored in the authorization data base:

| | | IDENTITY | ACCESS RIGHTS |
|---|---|---|---|
| #1 | TOKEN TYPE | access_id_USER | pos_access_rights |
| | DEF. AUTHORITY | KerberosV5 | local_manager |
| | VALUE | tom@ORG.EDU | FILE : read |

| | | IDENTITY | ACCESS RIGHTS |
|---|---|---|---|
| #2 | TOKEN TYPE | access_id_GROUP | pos_access_rights |
| | DEF. AUTHORITY | KerberosV5 | local_manager |
| | VALUE | admin@ORG.EDU | FILE : read,write |

| | | IDENTITY | ACCESS RIGHTS |
|---|---|---|---|
| #3 | TOKEN TYPE | access_id_USER | pos_access_rights |
| | DEF. AUTHORITY | KerberosV5 | local_manager |
| | VALUE | joe@ORG.EDU | FILE : write |

**Table 1.**

Credentials may have optional conditions associated with the granted rights. Assume the following credentials are stored in the security context associated with the user Tom.

Identity credential:

*access_id_USER kerberos.v5 tom@ORG.EDU*
`condition:` *time_window pacific_tzone 6am-7pm*

Group membership credential:

*access_id_GROUP kerberosV5 admin@ORG.EDU*
`condition:` *privilege:restricted*

Delegation credential:

`grantor:` *grantor_id_USER kerberosV5 joe@ORG.EDU*
`grantee:` *access_id_USER kerberosV5 tom@ORG.EDU*
`objects:` *doc.txt*
`rights:` *pos_access_rights local_manager FILE:write*
`condition:` *location local_manager *.org.edu*

Let's consider a request from a user Tom who is connecting from the *ORG.EDU* domain to write to the file *doc.txt* at 5pm.

In evaluating the EACL, the first entry does not grant the requested operation, however the second entry grants it. The evaluation function will then check the security context for the group `admin` membership credential. The proper credential is found, however, there is a condition `privilege:restricted`. This means that Tom can use this privilege only if logged in as an administrator. Evaluation continues. The third entry grants the requested operation. The evaluation function will look for a delegation credential for `tom@ORG.EDU` issued by `joe@ORG.EDU`. The appropriate delegation credential is found. The condition on location `*org.edu` is satisfied, so the requested access will be granted.

## 3.4 Generic Authorization and Access-control API (GAA API)

In this section we provide a description of the main GAA API routines.

### 3.4.1 GAA API functions

The `gaa_get_object_policy_info` function is called to obtain the security policy associated with the object.

- **Input**:
  - *Reference to the object to be accessed*. The identifier for the object is from an application-dependent name space, it can be represented as unique object identifier, or symbolic name local to the application.
  - *Pointer to application specific Authorization Database*.
  - *Upcall function for the retrieval of the object policy*. The application maintains authorization information in a form understood by the application. It can be stored in a file, database, directory service or in some other way. The upcall function provided for the GAA API retrieves this information and translates it into the internal representation understood by the GAA API.
- **Output**:
  - *Object policy handle*

The `gaa_check_authorization` function tells the application server whether the requested operations are authorized, or if additional application-specific checks are required.

- **Input**:
  - *Object policy handle, returned by* `gaa_get_object_policy_info`
  - *Principal's security context* (see section 3.5.1)
  - *Operations for authorization*. This argument indicates requested operations.
- **Output**:
  - `YES` (indicating authorization) is returned if all requested operations are authorized.
  - `NO` (indicating denial of authorization) is returned if at least one operation is not authorized.

  - `MAYBE` (indicating a need for application-specific checks) is returned if there are some unevaluated conditions and additional application-specific checks are needed, or if continuous evaluation of conditions is required.
  - *detailed answer* contains:
    - Authorization valid time period. The time period during which the authorization is granted is returned as condition to be checked by the application.
      Expiration time is calculated by the GAA API, based on:
      1. Time-related conditions in the object policy, e.g. EACL matching entries.
      2. Restrictions in the authentication and authorization credentials.
    - The requested operations are returned marked as granted or denied along with a list of corresponding conditions, if any. Each condition is marked as evaluated or not evaluated, and if evaluated marked as met, not met or further evaluation or enforcement is required. This tells the application which policies must be enforced.
    - Information about additional security attributes required. Additional credentials might be required from clients to perform certain operations, e.g. group membership or delegated credentials.

- `gaa_inquire_object_policy_info`

  This function allows the application to discover access control policies associated with the targeted object applied to a particular principal. It returns a list of rights that the principal is authorized for and corresponding conditions, if any. The application must understand the conditions that are returned unevaluated, or it must reject the request. If understood, the application checks the conditions against information about the request, the target object, or environmental conditions to determine whether the conditions are met. Actual enforcement of policies expressed through application specific conditions is the responsibility of the application and is outside of the scope of this paper.

### 3.4.2 GAA API Security Context

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

**Identity** Verified authentication information, such as principal ID for a particular security mechanism. To determine which entries apply, the GAA API checks if the specified principal ID appears in an EACL entry that is paired with a privilege for the type of access requested.

**Authorization Attributes** Verified authorization credentials, such as group membership, group non-membership, delegation credentials, and capabilities.

**Evaluation and Retrieval Functions for Upcalls** These functions are called to evaluate application-specific conditions, to request additional credentials, and to verify them.

## 4 Creation of the GAA API security context

Prior to calling the `gaa_check_authorization` function, the application must obtain the authenticated principal's identity and store it in the security context. This context may be constructed from credentials obtained from different mechanisms, e.g. GSS API, Kerberos, or others. This scenario places a heavy burden on the application programmer to provide the integration of the security mechanism with the application. A second scenario is to obtain the authentication credentials from a transport protocol that already has the security context integrated with it. For example, the application can call SSL or authenticated RPC. In this case, it is the implementation of the transport mechanism (usually written by someone other than the application programmer) which calls the security API requesting principal's identity.

The principal's authentication information is placed into the security context and passed to the GAA API. When additional security attributes are required for the requested operation, the list of required attributes is returned to the application, which may request them. Through the security context, the application may provide the GAA API with an upcall function for requesting required additional credentials. The credentials pulled by the GAA API are verified and added to the security context by the upcall function.

## 5 An Extended Example

To illustrate our approach we describe a simple Printer Manager application, where protected objects are printers. The Printer Manager accepts requests from users to access printers and invokes the GAA API routines to make authorization decisions, under the assumption that the administrator of the resources has specified the local policy regarding the use of the resources by means of EACL files. These files are stored in an authorization database, maintained by the Printer Manager.

## 5.1 Conditions

Administrators will be more willing to grant access to the printers if they can restrict the access to the resources to only users and organizations they trust. Further, the administrators may need to specify time availability, restrictions on resources consumed by the clients and accounting for the consumed resources. To specify these limits, the Printer Manager uses generic conditions, such as time, location, payment and quota. As an example of Printer Manager-specific condition, consider printer load, expressed as maximum number of jobs that may be in the queue.

## 5.2 Authorization Walk-through

Here we present an authorization scenario to demonstrate the use of the authorization framework for the case of printing a document. Assume Kerberos V5 is used for principal authentication. Assume that printer *ps12a* has the following ordered EACL shown in **Table 2.** stored in the Printer Manager authorization database.

| | | IDENTITY | ACCESS RIGHTS | CONDITIONS | |
|---|---|---|---|---|---|
| #1 | TOKEN TYPE | access_id_USER | pos_access_rights | time_window | printer_load |
| | DEF. AUTHORITY | KerberosV5 | local_manager | pacific_tzone | local_manager |
| | VALUE | joe@ORG.EDU | PRINTER : submit_print_job | 6AM-8PM | 20% |

| | | IDENTITY | ACCESS RIGHTS | |
|---|---|---|---|---|
| #2 | TOKEN TYPE | access_identity_GROUP | positive_access_rights | positive_access_rights |
| | DEF. AUTHORITY | KerberosV5 | local_manager | local_manager |
| | VALUE | operator@ORG.EDU | PRINTER : * | DEVICE : power_down |
| | TOKEN TYPE | access_identity_USER | | |
| | DEF. AUTHORITY | KerberosV5 | | |
| | VALUE | tom@ORG.EDU | | |

| | | IDENTITY | ACCESS RIGHTS | CONDITIONS | |
|---|---|---|---|---|---|
| #3 | TOKEN TYPE | access_id_ANYBODY | pos_access_rights | time_day | time_window |
| | DEF. AUTHORITY | none | local_manager | local_manager | pacific_tzone |
| | VALUE | none | PRINTER:view_printer_capabilities | sat-sun | 6AM-8PM |

**Table 2.**

Let's consider a request from user Tom who is connecting from the `ORG.EDU` domain to print a document on the printer *ps12a* at 7:30 PM.

When a client process running on behalf of the user contacts the Printer Manager with the request to `submit_print_job` to printer *ps12a*, the Printer Manager first calls `gaa_get_object_policy_info` to obtain a handle to the EACL of printer *ps12a*. The upcall function for retrieving the EACL for the specified object from the Authorization Database system is passed to the GAA API and is called by `gaa_get_object_policy_info`, which returns the EACL handle.

The Printer Manager must place the principal's authenticated identity in the security context to pass into the `gaa_check_authorization` function. This context may be constructed according to the first or second scenario, described in Section 8. If Tom is authenticated successfully, then verified identity credentials are placed into the security context, specifying Tom as the Kerberos principal `tom@ORG.EDU`.

Next, the Printer Manager calls the `gaa_check_authorization` function. In evaluating the EACL, the first entry applies. It grants the requested operation, but there are two conditions that must be evaluated.

The first condition is generic and is evaluated directly by the GAA API. Since, the request was issued at 7:30 PM this condition is satisfied. The second condition is specific. If the security context defined a condition evaluation function for upcall, then this function is invoked and if this condition is met then the final answer is `YES` (authorized) and detailed answer contains an authorization expiration time : 8PM (assume that authentication credential has expiration time 9PM), allowed operation `submit_print_job` and two conditions. Both conditions are marked as evaluated and met. During the execution of the task the Printer Manager is enforcing 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 second condition is marked as not evaluated and must be checked by the Printer Manager.

When additional credentials are needed, if the security context defines a credential retrieval function for the upcall, it is invoked. If the requested credential is obtained, then the final answer is `YES`. If the upcall function was not passed to the GAA API, the answer is `NO`.

## 6 Integration with alternative authentication mechanisms

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 and the unambiguous identification of principals in each, but it does not translate between heterogeneous authentication mechanisms.

We have integrated our distributed model for authorization with the Prospero Resource Manager (PRM), a metacomputing resource allocation system developed at USC. PRM uses Kerberos [2] to achieve strong authentication. PRM uses calls to the Asynchronous Reliable Delivery Protocol (ARDP) [16], 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 the principal's authentication information.

In addition, we have integrated the framework with the Globus Security Infrastructure (GSI), a component of the

Globus metacomputing Toolkit [18]. GSI is implemented on top of the GSS-API which allows the integration of different underlying security mechanisms. Currently, GSI implementation uses SSL authentication protocol with X.509 certificates.

Public key authentication requires consideration of the trustworthiness of the certifying authorities for the purpose of public key certification. Authentication is not based on the public key alone, since anybody can issue a valid certificate.

Certificates can comprise a chain, where each certificate (except the last one) is followed by a certificate of its issuer. Reliable authentication of a public key must be based on a complete chain of certificates which starts at an end-entity (e.g. user) certificate, includes zero or more Certification Authorities (CA) certificates and ends at a self-signed root certificate. A policy must be specified to validate the legitimacy of the received certificate chain and the authenticity of the specified keys. The following is an example of an EACL used for describing the Globus policy for what CAs are allowed to sign which certificates. The Globus CA can sign certificates for Globus or the Alliance. The Alliance CA can sign certificates for the Alliance.

`Token Type:` *access_id_CA*
`Defining Authority:` *X509*
`Value:` */C=US/O=Globus/CN=Globus CA*

`Token Type:` *pos_access_rights*
`Defining Authority:` *globus*
`Value:` *CA:sign*

`Token Type:` *cond_subjects*
`Defining Authority:` *globus*
`Value:` */C=us/O=Globus/* /C=us/O=Alliance/**

## 7 Groups and Roles

A group is a convenient method to associate a name with a set of subjects and to use this group name for access control purposes. The kind of subject (individual user, host, application or other group) composing the group is opaque to the authorization mechanism. A group server issues group membership and non-membership certificates.

In general, a principal may be a member of several groups. By default, a principal operates with the union of privileges of all groups to which it belongs, as well as all of his individual privileges.

Some applications adopt role-based access control. The concept of roles is not consistent across different systems. Several definitions of roles are present in the literature. In general, a role is named collection of privileges needed to perform specific tasks in the system. Role properties [4] include:

- A user can be a member of several roles
- Role can be activated and deactivated by users at their discretion.
- Authorizations given to a role are applicable only when that role is activated.
- There may be various constraints placed on the use of roles, e.g. a user can activate just one role at a time.

Shandu et. al. [10] view roles as a policy and groups as a mechanism for role implementation. We adopt this point of view. In our framework we implement different flavors of roles using the notion of group and a set of restrictions on granted privileges. Consider a role-based policy, which assigns users: Tom, Joe, and Ken role `Bank_Teller`. This role allows a legitimate user to perform deposit and withdraw operations on objects *account_1* and *account_2*. This policy may be easily expressed by our EACL framework:

1. Group `Bank_Teller` is defined which will include Tom, Joe, and Ken
2. The EACLs for objects *account_1* and *account_2* will contain the following entry:

   `Token Type:` *access_id_GROUP*
   `Defining Authority:` *X.509*
   `Value:` */C=US/O=Globus/CN=Bank Teller*

   `Token Type:` *pos_access_rights*
   `Defining Authority:` *pasific_coast_bank*
   `Value:` *ACCOUNT:deposit,withdraw*

In expressing role-based policy using groups, the issue of constraints on role activation and use should be addressed.

## 8 Clark-Wilson

The Clark-Wilson model [12] was developed to address security issues in commercial environments. The model uses two categories of mechanisms to realize integrity: well-formed transactions and separation of duty.

Our framework is designed to handle the Clark-Wilson integrity model. A possible way to represent a constraint that only certain trusted programs can modify objects is using `application:checksum` condition, where the checksum ensures authenticity of the application. Another way is using `application:endorser` condition, which indicates that a valid certificate, stating that the application has been endorsed by the specified endorser, must be presented.

Static separation of duty is enforced by the security administrator when assigning group membership. Dynamic separation of duty enforces control over how permissions are used at the access time [6]. Here are examples of EACL conditions specific to the Dynamic separation of duty:

- `privilege:restricted` Makes subject operate with the privilege of only one group at a time.
- `privilege:set_of_groups` Makes subject operate with the privilege of only specified groups at a time.
- `endorsement:list_of_endorsers` Concurrence of several subjects to perform some operation.

## 9 Lattice-based Policies

Our framework allows incorporation of Mandatory Confidentiality [14], Mandatory Integrity [15] models and their combination.

Mandatory policies govern access on the basis of classification of subjects and objects in the system. Objects and subjects are assigned security labels:

1. Confidentiality labels, e.g. Top_Secret/NASA, Sensitive/Department2
2. Integrity labels, e.g. High_integrity, Low_integrity
3. Single security labels for both confidentiality and integrity, e.g. Top_Secret/NASA, Unclassified. Assume that the first label denotes high integrity level, whereas the second one denotes low integrity level.

To prove eligibility to access an object, a subject has to present a valid credential, stating subject's security label.

All access rights are divided into read-class and write-class. Appropriate rules are applied to each class.

Generic conditions for read-class access rights:

a) `conf_read_equal:cofidentiality_label`

This condition specifies that a subject, wishing to get read-class access to the object has to have security clearance equal to the one, specified in the cofidentiality_label field.

b) `conf_read_below:cofidentiality_label`

This condition is used to enforce `read down` mandatory confidentiality rule. It specifies that a subject, wishing to get read-class access to the object has to have security clearance no less the one, specified in the `cofidentiality_label` field.

c) `integr_read_equal:integrity_label`

This condition specifies that a subject, wishing to get read-class access to the object has to have security clearance equal to the one, specified in the `integrity_label` field.

d) `integr_read_above:integrity_label`

This condition is used to enforce `read up` mandatory integrity rule. It specifies that a subject, wishing to get read-class access to the object has to have integrity clearance less or equal to the one, specified in the `integrity_label` field.

Similarly we define generic conditions for write-class access rights. Assume file *doc.txt* has classification Sensitive/Departmen1 and integrity label Medium, then EACL for this file can be specified as:

| | | IDENTITY | ACCESS RIGHTS | CONDITIONS | |
|---|---|---|---|---|---|
| #1 | TOKEN TYPE | access_id_ANIBODY | pos_access_rights | conf_write_above | integr_write_below |
| | DEF. AUTHORITY | none | system_manager | system_manager | system_manager |
| | VALUE | none | FILE : write | Sensitive/Deprt1 | Medium |
| | | | pos_access_rights | conf_read_below | |
| | | | system_manager | system_manager | |
| | | | FILE : read | Sensitive/Deprt1 | |

**Table 3.**

Note that in the example above, everybody in the distributed system can get read or write access to the file if a valid credential stating the appropriate security label attribute is presented. This poses a requirement that security labels be unique across different security domains. This may not be easily satisfied.

A possible way to restrict the scope of security labels to a particular administrative domain is to specify an additional condition such as location.

## 10 Conclusions

In this paper we presented a generic authorization mechanism that supports a variety of security mechanisms based on public or secret key cryptography. The mechanism is extensible across multiple applications supporting different operations and different kinds of protected objects. Alternative implementations may be chosen for underling security services that support the API. By extending the traditional ACLs and capabilities with conditions on authorized rights we are able to support a flexible distributed authorization mechanism, allowing applications and users to define their own access control policies either independently or in conjunction with centralized authorization and group servers. The problem of policy translation is addressed by using generic or application-specific evaluation functions. We are going to investigate the request and evaluation of additional credentials. The assumption that all relevant credentials are passed for evaluation contradicts privacy requirements. It might not be always desirable to reveal group membership and principal attributes up front. We have integrated our model with several applications.

## 11 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 EACL is specified according to the following format:

```
    eacl ::= {eacl_entry}
    eacl_entry ::=
access_id {access_id} pos_access_rights {condition}
{pos_access_rights {condition}} |
access_id {access_id} neg_access_rights

    access_id ::=
access_id_type def_authority value

    access_id_type ::=
"access_id_HOST" |
"access_id_USER" |
"access_id_GROUP" |
"access_id_CA" |
"access_id_APPLICATION" |
"access_id_ANYBODY"
```

A capability is defined according to the following format:

```
    capability ::=
grantor_id pos_access_rights {condition}
{pos_access_rights {condiction}}

    grantor_id ::=
grantor_id_type def_authority value

    grantor_id_type ::=
"grantor_id_HOST" |
"grantor_id_USER" |
"grantor_id_GROUP" |
"grantor_id_CA" |
"grantor_id_APPLICATION" |
"grantor_id_ANYBODY"

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

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

    condition ::=
condition_type def_authority value

    condition_type ::= alphanumeric_string

    def_authority ::= alphanumeric_string

    value ::= alphanumeric_string
```

## 12 Acknowledgments

This research was supported in part by the Information Technology Office of the Defense Advanced Research Projects Agency (DoD) under the Scalable Computing Infrastructure (SCOPE) Project, Contract No. DABT63-95-C-0095, Security Infrastructure for Large Distributed Systems (SILDS) Project, Contract No. DABT63-94-C-0034, Global Operating Systems Technology (GOST) Project, Contract No. J-FBI-95-204, Diplomat, Project Department of Energy Cooperative Agreement No. DE-FC03-99ER25397 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 the U.S. Army Intelligence Center and Fort Huachuca Directorate of Contracting, the Defense Advanced Research Projects Agency, the U.S. Government, or Xerox Corporation.

## References

[1] C. Neuman. Proxy-based authorization and accounting for distributed systems. *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh*, May 1993.

[2] C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–38, September 1994.

[3] T. Y. C. Woo and S.S. Lam. Designing a Distributed Authorization Service. *In Procedings IEEE INFOCOM '98*, San Francisco, March 1998.

[4] S. Jajodia, P. Samarati and V.S. Subrahmanian. A logical Language for Expressing Authorizations. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[5] 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.

[6] R. T. Simon and M. E. Zurko Separation of Duty in Role-Based Environments. *Computer Security Foundations Workshop*, June 1997.

[7] 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.

[8] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. *Proceedings of Network and Distributed System Security Symposium, San Diego, California*, March 1998.

[9] M. Blaze, J. Feigenbaum and J. Lacy. Decentralized Trust Management. in *Proc. IEEE Symp. on Security and Privacy, IEEE Computer Press, Los Angeles*, pages 164-173, 1996.

[10] R. S. Shandhu, E. J. Coyne, et al Role-Based Access Control: A Multi-Dimensional View. *Proc. of 10th Annual Computer Security Applications Conference*, December 5-9, pages 54-62, 1994.

[11] W. Shen and P. Dewan Access Control for Collaborative Environments. *Proc. of CSCW*, November, 1992, pages 51-58

[12] D. D. Clark and D. R. Wilson Non Discretionary Controls Commercial Applications. *Proc. of the IEEE Symposium on Security and Privacy*, pages 184-194, April 1997.

[13] S B. Lipner A Comparison of Commercial and Military Computer Security Policies *Proc. of the 1987 IEEE Symposium on Security and Privacy*, 1982.

[14] D. Elliott Bell and L. J. LaPadula *Secure Computer System: Unified Exposition and Multics. Interpretation, ESD-TR-75-306 (MTR-2997), The MITRE Corporation* Bedford, Massachusetts, July 1975.

[15] K. J. Biba Integrity Considerations for Secure Computer Systems, *The MITRE Corporation*, Bedford, MA, MTR-3153, 30 June 1975.

[16] N. Salehi, K. Obraczka and C. Neuman The performance of a reliable, request-response transport protocol. *Proceedings of the Fourth IEEE Symposium on Computers and Communications*, 6-8 July, 1999.

[17] Edited by I. Foster and C. Kesselman.
The GRID: Blueprint for a New Computing Infrastructure *Morgan Kauffman Publishers*, 1999.

[18] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kauffman Publishers, 1999.