The Specification and Enforcement of Advanced Security Policies

Tatyana Ryutov and Clifford Neuman
Information Sciences Institute
University of Southern California
{tryutov, bcn}@isi.edu

Abstract

In a distributed multi-user environment, the security policy must not only specify legitimate user privileges but also aid in the detection of the abuse of the privileges and adapt to perceived system threat conditions.

This paper advocates extending authorization policy evaluation mechanisms with a means for generating audit data allowing immediate notification of suspicious application level activity. It additionally suggests that the evaluation of the policies themselves adapt to perceived network threat conditions, possibly affected by the receipt of such audit data by other processes.

Such advanced policies assist in detecting and responding to intrusion and misuse and they allow more efficient utilization of security services, such as authentication, audit, and notification.

We present an authorization framework, which enables the representation and enforcement of advanced security policies. Our approach is based on expanding the policy evaluation mechanism with the ability to generate real time actions, such as checking the current system threat level and sending a notification.

1 Introduction and Motivation

As more and more enterprises make their critical information available on the Internet, whether only to employees or to end-customers, they are exposed to significant risks such as theft, fraud, and denial of service attacks. In general, the most significant consequences result from attacks within the system by otherwise legitimate users (or attackers posing as such users) performing unauthorized activities.

Detecting these kinds of attacks can require instrumenting applications to generate audit records based on activity that is only understood at the application layer.

In addition to having a means to detect attacks (the role of an intrusion detection system) it is essential to have well defined policies that indicate what to do under perceived attack conditions, or for that matter under suspicion of attack conditions so that data can be gathered to make an actual determination of whether an attack is present.

Countermeasures to such attacks must similarly be implemented at the application layers through enforcement of policies that can distinguish legitimate and illegitimate activities - a distinction that often requires application level knowledge.

While users might not be prevented from using resources to which they have legitimate access, protective measures, such as audit analysis along with the threshold control can be used to examine user actions. Consider an authorization policy: "Members of department D can access the printer P. If the number of print jobs created during the day is higher than 20, activate audit to log time, file and account names". In this policy the threshold is used to detect suspicious use of resources. An audit log can reveal that an individual is printing far more records than the average user, which could indicate the running of a covert business.

The policies themselves must automatically adapt to meet the changing security requirements in the event of possible intrusion while allowing users to operate in the changing environment. For example, consider authorization policy: "Tom can connect to host $malta.isi.edu$ if the system threat level is low (normal operational state). If the system threat level is medium (indicates suspicious behavior), Tom can connect only from a host within the administrative domain $isi.edu$. The connection duration time should not exceed 2 hours. If the system threat level is high (system is under attack), Tom can not connect."

Current access control systems are based on the premise that once a user is authorized to perform some operation, the access is granted unconditionally. This practice is not likely to detect the abuse of user privileges. To provide additional level of security checks, close monitoring of authorized actions may be necessary. Policies can be applied to controlling execution of the requested actions.

The points of the policy enforcement may include three time phases:

1. Before requested operation starts; to decide whether

this operation is authorized.

2. During the execution of the authorized operation; to detect malicious behavior in real-time (e.g., a user process consumes excessive system resources).

3. When the operation is completed; to activate post execution actions, such as logging and notification whether the operation succeeds/fails.

To protect sensitive and critical system resources in distributed environments, a system must be capable of supporting **advanced security policies**:

1. The policies must be adaptive [1] to accommodate changes in the security requirements and assist in detecting and responding to intrusion and misuse. To do so, the policies should indicate not only what activities are authorized, but also provide the means to detect abuse of user privileges. In particular, the policy should specify when audit records should be generated and allow for immediate notification.

2. Policy enforcement can be required at various time stages of the requested action. Thus, the policies should indicate when the policy has to be enforced.

The ability to enforce advanced policies has practical importance, for example, in computational Grids [5]. Grids are large-scale distributed computing environments that enable applications to use scientific instruments, computational and information resources that are managed by diverse organizations.

System administrators contributing their resources to a Grid will require assurance that the resources are adequately protected. In a Grid setting, the security requirements include:

1. User authentication.
   Authenticated user identity is used to determine who gains access to local resources [2].

2. Resource usage limits (quotas).
   A site-specific resource allocation policy specifies limits on the computational or storage resources to be consumed, such as CPU load, memory usage and disk space. The limits are taken into account when deciding whether to initiate the requested computation. Monitoring execution of the computation on a particular node must be supported to ensure that the process keeps strictly to the limits imposed by the local policy.

3. Accounting and payment.
   Owners of the resources may hold users accountable for the consumed resources. Accounting may include gathering information about executed computations and consumed resources. The accounting information can be used in payment models for remote service providers.

4. Audit.
   Audit can provide a means to help accomplish individual accountability and provide data to be analyzed by intrusion or misuse detection systems.

5. Intrusion and misuse detection.
   Grids are vulnerable to a large-scale malicious attacks that could cause disruption of the Grid services. Thus, it is essential for Grids to support detection and automatic response to intrusion attempts.

6. Event notification.
   Tools for intrusion detection and fault tolerance can be driven by event services. Alert-level notification messages permit cooperative responses. For example, notification about a computation that exceeds the quotas can signal ongoing denial of service attack. The adequate preventive measures can be taken if the attack is confirmed.

Authentication, authorization, audit, notification and intrusion detection systems are interrelated and should be used together to support effective system security.

The goal of this work is to design an authorization system that supports the advanced security policies.

2 Approach

An authorization policy regulates access to objects. An object is a target of requests and it has to be protected, e.g., critical programs, files, hosts and print jobs.

An access right (alternative words that we use are operation, action and permission) is a particular type of access to a protected object, e.g., read or write. Specific system events, such as restarting or shutting down the system, system log-in and log-off can be modeled as access rights associated with the system, where the system is the protected object.

A condition describes the context in which each access right is granted.

In our framework, a policy is represented as a set of conditions associated with the access right. All conditions must be satisfied in order to allow an operation to be performed on a target object [3].

[1] The term "adaptive" in this paper is used to indicate that the security policy to be enforced depends on the current state of the system, e.g., system load, system threat level or time of day (more restrictive organizational policy may be enforced during after hours).

[2] Mutual authentication may be required to prove the server identity to the user.

[3] Our framework supports negative rights. If all conditions associated with the negative right are met, the access is denied.

Traditional security systems lack adaptive security policies and enforcement mechanisms. In the non-adaptive setting, the set of policies is chosen in advance, before the access request is received. The adaptive policy enforcement mechanism chooses the appropriate set of policies during the course of computation based on the current system state.

Adaptive policy implementation requires either the reloading of the policy or changing the policy computation algorithms [3]. Both of these approaches are ineffective and not scalable.

Our approach avoids policy reloading and switching to the different policy evaluation mode:

1. The policy specification describes more than one set of disjoint policies.

2. The policy evaluation mechanism is extended with the ability to read and write system state. The implementation is based on read and write conditions that provide support for monitoring and updating internal system structures and their runtime behaviors.

With the extended policy evaluation mechanism, transition between the disjoint sets of policies is regulated automatically by reading the system state (e.g., the time of day, or system threat level). The downside of this approach is the requirement for more tedious and careful policy specification and dealing with the side effects of the policy evaluation.

The advanced policies are specified using different conditions that permit run-time adaptation in the event of possible security attacks. To enforce the advanced security policies we adopted the three-phase policy enforcement scheme. During each phase only the specified set of all conditions in the policy is evaluated.

2.1 Conditions

Here we list several of the more useful conditions [13] that assist in detecting and responding to intrusion and misuse and they allow more efficient utilization of security services, such as authentication, audit, and notification.

- **access identity**

  This condition specifies an authenticated access identity. If a policy does not require authenticated user identity, authentication steps can be ignored or deferred until the policy explicitly requests it. An example of a policy, which is not concerned with the identity is "anyone can read file A if $10 is paid".

- **strength of authentication**

  This condition specifies the authentication mechanism or set of suitable mechanisms for authentication. Strong user authentication method (e.g., Kerberos [14]) can be activated in response to suspicious behavior.

- **time**

  This condition specifies time periods for which access is granted.

- **location**

  This condition specifies location of the user. Authorization is granted to the users residing on specific hosts, domains, or networks.

- **payment**

  This condition specifies a currency and an amount that must be paid prior to accessing an object.

- **quota**

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

- **audit**

  This condition enables automatic generation of audit data in response to access requests. An audit record should include sufficient information to establish what event occurred and what caused the event.

- **notification**

  This condition enables automatic generation of notification messages (alerts) in response to access requests. Specifies the receiver and the notification method.

- **threshold**

  This condition specifies allowable threshold.

- **system threat level**

  This condition specifies the system threat level.

Failure of some of these conditions may signal suspicious behavior. For example, access is requested at unexpected times or unusual locations, violations of user quotas, repeated failure of access attempts and exceeding a threshold. Some conditions can trigger defensive measures in response to perceived system threat level. For example, impose a limit on resource consumption, advanced payment for the allocated resources or increased auditing. In the case of insider misuse (particularly if the intruder's identity has been established) it may be appropriate to let the attacks continue under special conditions. For example, it may be desirable to initiate data collection mechanisms to gather detailed information about user activities that could serve as evidence for possible prosecutions.

The combination of conditions of different types can be used to fine tune audit and notification services. The audit detail and number of alarms should be sensitive to the system threat profile. For example, low system threat level should result in reduced alarm level and amount of generated audit data. It should also depend on the sensitivity of the requested operation and target object.

2.1.1 Evaluation of Conditions

Note that in the implementation, some of these conditions might have side effects. For example, evaluation of **payment** and **quota** conditions reduce a balance. Evaluation of **notification** condition results in sending a message, which is useful in audit.

Unfortunately, side effects complicate the system. Ignoring the side effects might cause problems when the side effects create a feedback loop, for example, when an audit record triggers a network threat detection which affects the evaluation of subsequent policies, or where payment affects quotas which affects the ability to perform other operations (once one runs out of money).

Another problem caused by the side effects, is possible inconsistency of the authorization result. For example, consider a policy "Tom can shut down host H only if a notification is sent (`notification`) and system threat level is low (`system_threat_level:low`)". Assume that the current system threat level is low. Assume that the notification about Tom shutting down the host triggers high system threat level (this may indicate attempted denial of service attack). There are two ways to evaluate the conditions: first `system_threat_level:low` then `notification`. This evaluation order results in access grant. Another way is to evaluate `notification` condition first then `system_threat_level:low`. This evaluation order results in the denial of the access.

All side effects of the condition evaluation are recorded in the corresponding system variables. At the lowest level, a system variable is an abstraction for bits or bytes in the system that change as the result of system execution. For example, to model a system variable affected by the evaluation of the **notification** condition (a message must be sent), we need better level of abstraction. Thus, a system variable is an abstract notion of a system entity that represents a data item, e.g., a file, a message or a record in a database. Each system variable has a name and a value.

We assume that there exists a set of software components S. Each software component s $(s \in S)$ can access system variables of particular type. For example, a system variable, which represents a file is accessed by a file system. A system variable, which represents a notification is accessed by a notification protocol, or a transport protocol, such as e-mail or http.

We assume that each software component s has abstract $Read$ and $Write$ operations as a part of its functionality. The read operation $s.Read(X)$ returns the value of the system variable X. The write operation $s.Write(X, v\_new)$ assigns a new value v_new to the system variable X.

2.1.2 Read and Write Conditions

At the conceptual level, all conditions can be categorized as:

- Conditions that require reading some system variable and comparing it with the information specified in the policy. For example, evaluation of the **time** condition requires obtaining current time and checking if it fits into the time interval specified in the policy. We call this category of conditions **read conditions**. A read condition is represented as $XopP$, where X is the name of a system variable, P is a constant and op is the operation (e.g., $=$, $\neq$, $<$, $>$) to be performed on the value of the system variable X and the constant P. In implementation, this value maybe either obtained from the request or read using the $s.Read(X)$ operation during the condition evaluation.
- Conditions that require writing some information (e.g., audit) or initiating some action (e.g., notification). We call this category of conditions **write conditions**. A write condition is represented as $Xnew\_value$, where X is the name of the system variable and new_value is the new value to be assigned.

An obvious relationship between the read and write conditions is if one condition requires reading of a system variable, which is written by the other condition. In our framework, the condition evaluation process is totally ordered. The order has to be assessed before condition evaluation starts. Determining the correct order of the conditions in the policy statement is an important issue. Human judgment is a necessary component in this process. We feel that the function of defining the condition order can be best served by having the policy officer chose a meaningful condition order. In particular, whether the **write conditions** must be evaluated before the **read conditions**. The goal of the system is to faithfully implement the given organizational security policy.

2.1.3 Pre-, Mid-, Post- and Request-result Conditions

An authorization policy may specify conditions that must be satisfied before, during or after the access right is exercised. Furthermore, evaluation of some conditions must be activated only if the authorization request is granted (or denied).

Thus, all conditions are classified as:

- **pre-conditions** specify what must be true in order to grant the request. This means that the requested operation is allowed to be executed on the target object. If any of the pre-conditions fails, authorization is denied.
- **request-result conditions**
  These conditions must be activated whether the authorization request is granted or whether the request is denied.
- **mid-conditions** specify what must be true during the execution of the requested operation. The mid-conditions can be used for the protection of the critical operations and resources. The mid-conditions allow for real time active monitoring of the operation execution and response. If any of the mid-conditions fails, the operation execution must be affected. The countermeasures are defined in the response methods of the target object. Aggressive responses may include direct countermeasures, such as closing the connections or suspending the processes. This is important to enforce counter measures against serious attacks. For example, a processes consuming excessive system resources (CPU time, memory, and disk space) may indicate impending denial of service attack. More passive responses may include the activating of integrity-checking routines to verify the operating state of the target.

  The mid-conditions that we consider in our framework are limited to a set of thresholds, such as duration of connection, CPU and memory usage and severity metrics (e.g., current system threat level).
- **post-conditions** specify what must be true on the completion of the operation execution. The post-conditions can be specified in two ways:

  1. The post-conditions that are activated only if the requested operation succeeds. These conditions are useful to correctly implement the enforcement of, for example, the payment/quota constraints.

     Here are some examples of the policies with post-conditions:

     "A user must pay $1 to read a file. The money must be withdrawn from the user account only after successful file access."

     In this policy, the **payment** condition must be implemented as a post-condition. If the file read fails for technical reasons (the server crashes in the middle of the read operation), the payment condition is not activated and the user does not lose his money.

     "A user is allowed to access file A only once."

     Similarly, the **quota** condition in this policy must be implemented as a post-condition to ensure that the user can access the file at least once.
  2. The post-conditions that are activated only if the requested operation fails. For example, failure of critical operations, such as system shut down may indicate denial of service attack and require immediate notification.

The post-conditions along with the request-result conditions are useful to fine tune audit and notification services.

2.2 The Three-Phase Policy Enforcement

The enforcement of the advanced security policies is partitioned into three successive phases.

1. Phase one: access control.
   The pre- and request-result conditions are evaluated during this phase and the decision to grant or deny access to the requested object is made.

2. Phase two: execution control.
   The access to the target object is granted, the requested operation is started and the mid-conditions are evaluated during this phase. This phase allows the controlled execution of the requested operation.

3. Phase three: post-execution actions.
   The post-conditions are evaluated during this phase. The specified actions are performed after the operation is finished. We do not call this phase "post-execution control", since neither failure nor success of a post-execution action can affect either access decision, or operation execution.

3 Implementation

In this section we present the overview of our implementation approach.

3.1 Policy Representation

The policy language that we implemented is called Extended Access Control List (EACL). The EACL is a simple policy language designed to describe user-level authorization policy. An EACL is associated with an object (or a group of objects) to be protected and specifies positive and negative access rights with optional set of associated conditions.

A **condition block** defines a conjunction of a totally ordered set of conditions. Conditions are evaluated in the order they appear within a condition block[4].

An **EACL entry** consists of a positive or negative access right and four condition blocks: a set of pre-conditions, a set of request-result conditions, a set of mid-conditions and a set of post-conditions. Note that a condition block can be empty. If all condition blocks in an EACL entry are empty, the right is granted unconditionally. An example of a practical policy with empty condition blocks is: "anyone can read file $index.html$".

An **EACL** consists of an ordered set of disjunctive EACL entries. An EACL representation supports disjunction and conjunction of conditions to activate different control modes.

An EACL is equivalent to disjunctive normal form consisting of a disjunction of conjunctions where no conjunction contains a disjunction. For example, a policy "Tom or Joe can read file A only if they connect from *.isi.edu domain" can be represented by an EACL (attached to the file A) with two EACL entries:
"positive access right: read, pre-conditions: Tom, *.isi.edu"
"positive access right: read, pre-conditions: Joe, *.isi.edu".

More precise EACL syntax and an example are given in the Appendix.

Evaluation of an 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 which already have been examined take precedence over new authorizations.

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.

3.2 Generic Authorization and Access-control API(GAA-API)

The GAA-API provides a general-purpose execution environment in which EACLs are evaluated. Next we provide a brief description of the main GAA-API functions.

The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policy associated with the object. It takes the target object and authorization database as input and returns an ordered list of EACLs.

The application maintains authorization information in a form understood by the application. It can be stored in a file, database, and directory service or in some other way. The application-specific callback function provided for the GAA-API retrieves the policy information and translates it into the internal representation understood by the GAA-API. Currently the policy is written at the object level, the call-back function must collect all the per object policies and order them by priority. How the policies are stored and retrieved is opaque to the GAA-API and is not reflected in the EACL.

The resulting policy that is passed to the GAA-API for evaluation represents the combination of several policies possibly from different domains and individual users of the system. The specific mechanism for retrieving the policies is passed as a call-back function.

The GAA-API provides a mechanism to register a particular policy retrieval call-back function. Currently this is done using a configuration file.

The structure of the policy domains that contribute the policies is not specified explicitly in our framework. Only the hierarchical relationship (priority of the policy) among the domains is taken into consideration. Our current implementation supports two level policy specification: first, system-wide policies are retrieved and placed in the beginning of the list of policies. Then the local policies are retrieved and are added to the list. Thus, system-wide policies implicitly have higher priority than local policies.

The $gaa\_check\_authorization$ function checks whether the requested right is authorized under the specified policy. This function takes the retrieved policy (an ordered list of EACLs), requested access right and contextual information as input. The contextual information is matched to the requirements, specified in the conditions of the relevant EACL entries (only the EACL entries where the the requested right appears are evaluated). For example, this information can be represented by a set of credentials, e.g., an X.509 identity certificate. The output lists all matching policy rights and associated conditions, with flags set to indicate whether each condition was evaluated and/or met. If the access is granted, the output includes the time period for which the result is valid.

$gaa\_execution\_control$ performs policy enforcement during operation execution. This function checks whether the mid-conditions associated with the granted access right are met.

$gaa\_post\_execution\_actions$ performs policy enforcement after the operation completes. This function enforces the post-conditions associated with the granted access.

A policy statement may specify several conditions of different types. For example: "Tom can read file A only between 9am and 6pm". This policy defines two pre-conditions: **access identity** and **time**. Both conditions are **read conditions** (there are two system variables to be read: user access identity and current time).

The GAA-API supports registering condition evaluation functions for different condition types.

[4] The total order property is important to deal with possible side effects caused by the condition evaluation.

The configuration file lists concrete functions that implement the conditions. The file is read at the GAA-API initialization time and the functions are registered with the specific conditions. In our policy example we define two functions: one to check the access identity and the other one to check the time. The read vs. write distinction shows up implicitly in the condition type. A condition evaluation function registered with a condition type knows whether the condition is **read** or **write**. It then parses the condition value and calls the concrete functions that implement the abstract $Read$ and $Write$ operations described in Section 2.1.1. The system variables manipulated by the $Read$ and $Write$ operations, as well as the operations themselves can be ether local or remote. However, our framework requires that the $Read$ and $Write$ operations must be implemented as atomic actions. The GAA-API is structured to support the addition of modules for evaluation of new conditions.

The $gaa\_check\_authorization$, $gaa\_execution\_control$ and $gaa\_post\_execution\_actions$ functions return the evaluation status $T/F/U$.

This status is obtained during the evaluation of conditions in the relevant EACL entries:
T indicates that all conditions are met;
F indicates that at least one of the conditions fails;
U indicates that none of the conditions fails but there is at least one condition that is left unevaluated.

Uncertainty U is introduced into our framework by lack of adequate information to evaluate the condition. For example, a condition may depend on an event that has yet to happen. This means that the value of the system variable returned by the implementation of the abstract $s.Read(X)$ operation is undefined. Another source of uncertainty is inability to find the corresponding condition evaluation function, for example if the function ($s.Read(X)$ or $s.Write(X, v\_new)$) is not implemented or not registered with the GAA-API. Sometimes, it is convenient to return some of the conditions unevaluated for further evaluation by the calling application.

3.3 The Policy Enforcement Process

The GAA-API returns three status values to describe policy enforcement process:

1. authorization status S_a.
   Indicates whether the request is authorized (T), not authorized (F) or uncertain (U).
2. mid-condition enforcement status S_m.
   Indicates the evaluation status of the mid-conditions ($T/F/U$).
3. post-condition enforcement status S_p.
   Indicates the evaluation status of the post-conditions ($T/F/U$).

Initially the status values are set to U.

1. The access control phase starts with receiving a request to access an object, requested type of access and contextual information.
2. First, the $gaa\_get\_object\_policy\_info$ function is called to obtain the security policy associated with the object. If no relevant policy was found, the authorization status is set to F and the request is rejected.

   Next the $gaa\_check\_authorization$ function is called to evaluate pre- and request-result conditions. If there are no pre-conditions (this means that the requested right is granted unconditionally), the authorization status is set to T. Otherwise, the pre-conditions are evaluated and the result is stored in the authorization status S_a.

   If the request-result conditions are present in the policy, the conditions are evaluated and the intermediate result is stored in variable X. The conjunction of the X and S_a is stored in the authorization status S_a. If authorization is not granted ($S_a \neq T$), the request is rejected.
3. The execution control phase consists of starting the operation execution process and calling the $gaa\_execution\_control$ function.

   If mid-conditions are found, the conditions are evaluated. Some mid-conditions are evaluated just once [5], other mid-conditions are evaluated in a loop until either the operation finishes or any of the mid-conditions fails. In the latter case, the operation execution is suspended and the reactive actions are started. The mid-conditions can be returned unevaluated to be enforced by application. The result is stored in S_m.
4. During the post-execution action phase the $gaa\_post\_execution\_actions$ function is called. The operation execution status (indicating whether the operation succeeded/failed) is passed to the $gaa\_post\_execution\_actions$. If no post-conditions are found, the S_p is set to T, otherwise the post-conditions are evaluated and the result is stored in S_p.

4 Related Work

The work by Huang and Shan [8] describes a SQL-like policy definition language. The policy enforcement process allows refining of the initial authorization request (request enhancement) and suggesting alternatives (request rewriting) if the requested resource is unavailable. These actions are performed by the policy enforcement mechanism before

[5] E.g., locking a file to place a hold on user account.

submitting the actual resource retrieval request to the resource manager. This approach is different from ours in that:

1. The point of the policy enforcement is at the creation of the resource request (based on the enhancement/rewriting of the initial request), which complies with existing policies. Then the resource is retrieved without any further checks. In our framework, the request is checked against the policies and is denied/granted or uncertain. No request modifications exist.

2. The approach has a limited condition representation model that does not support side effects.

The Policy Maker system described in the papers by Blaze et al. [1], [2] focuses on construction of a practical algorithm for a determining trust decisions. Policies and credentials encode trust relationships among the issuing sources.

In Policy Maker's terminology, "proof of compliance question" asks if the request q, supported by a set of credentials complies with a policy P. This is equivalent to the authorization question that we consider in our work: "is request q authorized by the policy P (in our model the credentials are contained in the request)". Their approach, however, is different from ours.

In our approach, the information passed to the authorization engine with the authorization request is used to evaluate conditions in the relevant policy statements. The order of condition evaluation is important.

The Policy Maker system is based on the logic programming approach. The goal is to infer the desired conclusion from given assumptions in a computationally viable manner. In Policy Maker, the credentials and policy (called assertions) are used collectively to compute a proof of compliance. The assertions can be run in an arbitrary order and produce intermediate results that then can be fed into other assertions.

Hayton and colleagues [7] proposed a role-based access control system called OASIS. OASIS services specify policy for role activation using Role Definition Language (RDL) that is defined in terms of axioms in proof system. These axioms are used to prove user's eligibility to enter a set of roles.

The policy for each set of services is specified at administrative domain level, with service level agreements between domains. The role names are local to each service. A role can be specified as being permitted only for those who can prove membership of other roles issued by this and other services. The services are responsible for issuing certificates, verifying their validity and notifying other services about the certificate state changes. A policy defines a set of conditions under which a user can activate a role. Condition evaluation is achieved by presenting a corresponding certificate. The role revocation is accomplished through membership conditions. Some of the membership conditions must continue to hold while the role remains active. If any of the membership conditions associated with the activated role fails, the role is deactivated. In some sense, the OASIS membership conditions are similar to our mid-conditions that must hold during operation execution.

RDL is not as generic and expressive as our approach and not as well suited to representing complex access control policies and those that include mandatory access control.

Policies, representable in Policy Maker and RDL, are restricted to the set of policies which do not produce side effects, resulting in change of the system state.

Ponder [4] is an object-oriented policy specification language that is suted for role-based access control policies, as well as general-purpose management policies. Ponder is targeted for different types of policies, including obligations, authorizations, delegation and filtering policies, and grouping these policies into aggregate structures. The obligation policies, for example, specify what actions (e.g., notification or logging) are carried out when specific events occur within the system. To some extent, the request-result and post-conditions in our framework serve a similar purpose. However, there are several significant differences between Ponder's and our approaches. First, in our framework all security requirements are expressed in a single policy structure, whereas in the Ponder approach authorization and obligation policies can be specified independently. These can lead to conflicts between the two policy types. Second, the policy in our framework is enforced by the same access control mechanism. The three-phase policy enforcement model allows for parts of policy (particular conditions) to be enforced at different times. In contrast, the Ponder uses a separate enforcement mechanism for each policy type.

Finally, the Ponder obligation policies are triggered by system events whereas in our framework the actions are triggered by other conditions in the same policy, such as threshold or system threat level.

Minsky and Ungureanu [11], [12] define the policy in terms of messages that only a restricted set of agents is permitted to exchange. Furthermore, the message exchange is controlled by a set of rules that is included in the policy. The policy enforcement mechanism is based on a set of trusted agents that interpret the rules and enforce them by regulating the message exchanges and the effect that the messages have on the control state (attributes and permissions) of the participating agents.

The ability to communicate and change the state resembles our concept of the read and write conditions. Our approach is different in that the "state" has a wider meaning. It includes all security-relevant information about real world which is representable in a computer system, e.g., bank account balance, temperature and user identity. Another difference is that the reading and writing of the state is based on

the ordered synchronous evaluation of the conditions, rather than controlled message exchange.

Jajodia et al. [9] have proposed a logical language for the specification of authorizations. The concerns addressed in this work are orthogonal to the ones in this paper. In particular, they focus on modeling conflict resolution, integrity constraint checking and derivation rules (that derive implicit authorizations from explicit ones), while our work focuses on the representation and enforcement of authorization policies enhanced with detection and management of security violations.

Summary of the research of audit-based intrusion and misuse detection is given by Lunt [10]. Sandhu and Samarati [17] discuss authentication, access control and intrusion detection technologies and suggest that combination of the techniques is necessary in order to build a secure system.

5 Conclusions and Future Work

Traditional authorization mechanisms check whether a user is acting within prescribed parameters and will not detect abuse of privileges. Advanced policies can conditionally generate audit records and in limited ways can react to state generated by intrusion detection engines based on observation of the audit records. Such policies can also adapt the level of detail of the audit records generated until an intrusion detection engine notices that something is amiss, though not necessarily what it is. Such policies can also adapt the applied authentication policies to require more information from a user when suspicious activity has been detected.

In this paper we presented an authorization framework that enables the specification and enforcement of advanced authorization policies.

The GAA-API implementation is available at
http://www.isi.edu/gost/info/gaaapi/source.
For further details about the authorization model see [16]. For more information about the GAA-API see [15].

The GAA-API has been integrated with several applications, including ssh and Globus Security Infrastructure [6]. Currently we are integrating the GAA-API with FreeS/WAN IPsec.

There are some aspects of distributed policy evaluation and enforcement that do not fit well within the framework. In the current framework we assume that conditions are evaluated consecutively and that authorization requests do not overlap. These two assumptions enable us to concentrate on a single condition evaluation at a time and, therefore, avoid the problem of coordination of multiple condition evaluation processes.

This results in inefficient policy evaluation process and leads to systems that cannot scale to large numbers of objects. Our current approach may be appropriate for some client-server applications, where the server is an autonomous agent, in complete charge of its resources. The server maintains the security policy and is responsible for the policy evaluation. Some distribution of the policy evaluation process can be achieved through the condition evaluation function implemented as, for example, an RPC call that is performed synchronously. However, this approach is not suitable for truly distributed architectures where a set of servers implement the policy and the policy evaluation processing can be distributed over several servers. Each server is responsible for enforcing of a part of the whole access control policy.

The future directions for this research include exploring extensions to the framework that could encompass these issues.

6 Appendix

We use the Backus-Naur Form to denote the elements of our policy language. 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 ::= pos_access_right conditions |
neg_access_right conditions
pos_access_right ::= "pos_access_right"
def_auth value
neg_access_right ::= "neg_access_right"
def_auth value
conditions ::= pre_conds mid_conds rr_conds
post_conds
pre_conds ::= {condition}
mid_conds ::= {condition}
rr_conds ::= {condition}
post_conds ::= {condition}
condition ::= cond_type def_auth value
cond_type ::= alphanumeric_string
def_auth ::= alphanumeric_string
value ::= alphanumeric_string
```

`cond_type` defines the type of condition, e.g., access identity or time.

`def_auth` indicates the authority responsible for defining the value within the `cond_type`, e.g., Kerberos.

`value` is the value of condition. Its semantics is determined by the `cond_type` field. The name space for the value is defined by the `def_auth` field.

It should be pointed out that the EACL language description presented here is not complete. Our current framework supports flexible policy composition model. The discussion of this issue is beyond the scope of this paper.

Next we present an example of an EACL that governs access to a host.

Entry 1 specifies that Tom can not login to the host.

Entries 2 and 3 mean that logins from the specified IP address range are permitted, using either X509 or Kerberos for authentication if the number of previous login attempts during the day does not exceed 3. If the request fails, the number of the failed logins for the user should be updated. The connection duration time must not exceed 8 hours.

Entry 4 means that anyone, without authentication, can check the status of the host if he connects from the specified IP address range.

Entry 5 specifies that host shut downs are permitted, using Kerberos for authentication. If the request succeeds, the user ID must be logged. If the operation fails, the sysadmin must be notified by e-mail.

```
    # EACL entry 1
neg_access_right test host_login

pre_cond_access_id KerberosV.5 tom@ORGB.EDU
    # EACL entry 2
pos_access_right test host_login

pre_cond_location IPsec 10.1.1.0-10.1.200.255
pre_cond_access_id X509
"/C=US/O=Trusted/OU=orgb.edu/CN=partnerB"
pre_cond_threshold local ≤3failures/day/failed_log/
rr_cond_update_log local on:failure/failed_log/info:userID
mid_cond_duration local ≤8hrs
    # EACL entry 3
pos_access_right test host_login

pre_cond_location IPsec 10.1.1.0-10.1.200.255
pre_cond_access_id KerberosV.5 partnerb@ORGB.EDU
pre_cond_threshold local ≤3failures/day/failed_log/
rr_cond_update_log local on:failure/failed_log/info:userID
mid_cond_duration local ≤8hrs
    # EACL entry 4
pos_access_right test host_check_status

pre_cond_location IPsec 10.1.1.0-10.1.200.255
    # EACL entry 5
pos_access_right test host_shut_down

pre_cond_access_id KerberosV.5 trusted@ORGA.EDU
rr_cond_audit local on:success/info:userID
post_cond_notify local email/to:sysadmin/on:failure
```

7 Acknowledgement of Sponsorship

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: (1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DEFCN); (2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); (3) J-FBI-95-204, Global Operating System Technologies Project (GOST); (4) DE-FC03-99ER25397, Diplomat project; and (5) HE1254-97, XAUTH Project.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

8 Disclaimer

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy, the U.S. Government or the Xerox Corporation.

References

[1] M. Blaze, J. Feigenbaum and J. Lacy.
Decentralized Trust Management.
Proceedings IEEE Symposium on Security and Privacy, IEEE Computer Press, Los Angeles, pages 164-173, 1996.

[2] M. Blaze, J. Feigenbaum and M. Strauss.
Compliance Checking in the Policy Maker Trust Management System. *In Proceedings of the Financial Cryptography '98*, *Lecture Notes in Computer Science*, volume 1465, pages 254-274.

[3] M. Carney and B. Loe.
A Comparison of Methods for Implementing Adaptive Security Policies. *In Proceedings of the 7th USENIX Security Symposium*, pages 1-14, January, 1998.

[4] N.Damianou, N. Dulay, E. Lupu and M. Sloman.
The Ponder Policy Specification Language. *In Proceedings of the Workshop on Policies for Distributed Systems and Networks*, Springer-Verlag LNCS 1995, pages 18-39, Bristol, UK, January, 2001.

[5] I. Foster and C. Kesselman, editors.
The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1999.

[6] I. Foster and C. Kesselman.
Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, Summer 1997.

[7] R. J. Hayton, J. M. Bacon and K. Moody.
OASIS: Access Control in an Open, Distributed Environment.
Proceedings of the IEEE Symposium on Security and Privacy, pages 3-14, Oakland, CA, May 1998.

[8] Y. Huang and M. Shan.
Policies in a Resource Manager of Workflow Systems: Modeling, Enforcement and Management. *Hewlett Packard Lab Software Technology Laboratory Technical Report HPL-98-156*, September, 1998.

[9] 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.

[10] T. F. Lunt.
A Survey of Intrusion Detection Techniques, *Computers and Security*, volume 12, pages 405-418, June 1993.

[11] N. Minsky and V. Ungureanu.
Unified Support for Heterogeneous Security Policies in Distributed Systems. *In 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[12] N. Minsky and V. Ungureanu.
Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. *In ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol 9, No 3, pages 273-305, July 2000.

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

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

[15] T. V. Ryutov and B. C. Neuman.
Representation and Evaluation of Security policies for Distributed system Services. *In Proceedings of the DARPA Information Survivability Conference and Exposition*, January 2000. Hilton Head, South Carolina.

[16] T. V. Ryutov and B. C. Neuman.
The Set and Function Approach to Modeling Authorization in Distributed Systems. *In Proceedings of the Workshop on Mathematical Methods and Models and Architecture for Computer Networks Security*, May 2001, St. Petersburg Russia.

[17] R. Sandhu and P. Samarati.
Authentication, Access Control, and Intrusion Detection.
The Computer Science and Engineering Handbook, pages 1929-1948, CRC Press, 1997.