Dynamic Authorization and Intrusion Response in Distributed Systems

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

Abstract

This paper[1] presents an authorization framework for supporting fine-grained access control policies enhanced with light-weight intrusion/misuse detectors and response capabilities. The framework intercepts and analyzes access requests and dynamically adjusts security policies to prevent attackers from exploiting application level vulnerabilities.

We present a practical, flexible implementation of the framework based on the Generic Authorization and Access Control API (GAA-API) that provides dynamic authorization and intrusion response capabilities for many applications. To evaluate our approach, we integrated the API with several applications, including Apache web server [12], sshd and FreeS/WAN IPsec for Linux. This paper demonstrates the integration of the GAA-API into ssh daemon. By integrating the GAA-API into sshd, the ssh server can support fine-grained authorization policies, dynamic policy update, and application level intrusion detection and response. The server can also enforce policies with additional functionalities, e.g., time- and location-based controls. Our experiments showed that the required integration effort was moderate, and that the performance impact on the ssh server was negligible.

1 Introduction and Motivation

As more and more enterprises make their critical information available on the Internet, whether only to employees or to 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.

[1] Portions reprinted, with permission, from T. V. Ryutov and B. C. Neuman. The Specification and Enforcement of Advanced Security Policies. In the Proceedings of the Conference on Policies for Distributed Systems and Networks (POLICY 2002). ©2002 IEEE.

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

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.

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.

Access control policies can assist in the application-based category of intrusion detection, which monitors critical applications. Traditional access control policies simply specify whether the access is granted or whether the request is denied. A new policy specification approach with intrusion detection in mind (in addition to defining actions that are and are not permitted) will identify specific application level events that constitute malicious or suspicious activities. Furthermore, such policies will specify the countermeasures to be taken to respond to the suspected or detected attacks.

We apply dynamic authorization techniques to support fine-grained access control and application level intrusion/misuse detection and response capabilities.

Our approach is based on specifying access control policies extended with the capability to identify (and possibly classify) intrusions and respond to the intrusions in real time. The Generic Authorization and Access Control API (GAA-API) is a generic interface which may be used to enable such dynamic authorization and intrusion response capabilities for many applications. The API supports three policy enforcement 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.

This paper demonstrates the integration of the GAA-API into ssh daemon. By integrating the GAA-API into sshd, the ssh server can support fine-grained authorization policies, dynamic policy update, and application level intrusion detection and response. The server can also enforce policies with additional functionalities, e.g., time- and location-based controls. Our experiments showed that the required integration effort was moderate, and that the performance impact on the ssh server was negligible with relatively small configuration and policy files.

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 and hosts. An *access right* (alternative words that we use are operation and action) 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 or denied.

In our framework, a policy is represented as a set of conditions associated with a positive or negative access right. If all conditions associated with a positive right are met, the access to a target object is granted. If all conditions associated with a 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.

Usually, 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 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 adaptive policies are specified using different conditions that permit run-time adaptation in the event of possible security attacks. To enforce the adaptive 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 [10] 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.

- **strength of authentication**

  This condition specifies the authentication mechanism or set of suitable mechanisms for authentication. Strong user authentication method (e.g., Kerberos [11]) 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. The condition value 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** condition reduces a balance. Evaluation of **notification** condition results in sending a message, which is useful in audit.

A positive aspect of the side effects is the ability to update system behavior at run time (e.g., generating audit records and reconfiguring firewall rules). Such dynamic techniques will ensure that policies applied to system services adapt to perceived system threat profile, thereby increasing system protection.

Unfortunately, side effects complicate matters. There are two particular difficulties in reasoning about policies enforced in the dynamic authorization environment.

First, the side effects might cause problems when the side effects create a feedback loop, e.g., when payment affects quotas which affects the ability to perform other operations (once one runs out of money).

Second, policy rules can include both environmental conditions and actions that change the conditions. For example, an audit condition may trigger a network threat detection condition which affects the evaluation of subsequent conditions in the policy. Therefore, the consistency and correctness of the access control desicions may depend on the condition evaluation order.

In our current framework, the condition evaluation process is totally ordered. The order has to be assessed before condition evaluation starts. Determining the evaluation ordering is currently done by a policy officer.

We recognize that the function of defining the condition order can be best served by an automated tool to ensure policy correctness and consistency and to ease the policy specification burden on the policy officer. See section 8 for further discussion.

2.1.2 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 whether the access is granted or whether the request is denied. Thus, all conditions are classified as:

- **pre-conditions** specify what must be true in order to grant or deny the request.

- **request-result conditions** must be activated whether the access 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** are used to activate post execution actions, such as logging and notification whether the operation succeeds or whether the operation fails. 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 adaptive 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 we implemented is called Extended Access Control List (EACL). The EACL is a simple language designed to describe user-level security policies that govern access to protected resources, identify threats that may occur within application and specify intrusion response actions. 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 that describe the context in which each access right is granted or denied.

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[2].

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".

3.1.1 EACL Syntax

We use the Backus-Naur Form to denote the elements of our EACL language. Curly brackets, $\{\}$, surround items that can

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

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 pre_cond_block
pos_access_right ::= "pos_access_right"
def_auth value
neg_access_right ::= "neg_access_right"
def_auth value
conditions ::= pre_cond_block mid_cond_block
rr_cond_block post_cond_block
pre_cond_block ::= {condition}
mid_cond_block ::= {condition}
rr_cond_block ::= {condition}
post_cond_block ::= {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.

Note that the EACL syntax allows only the pre-conditions to be associated with a negative right. This is because an EACL entry with a negative right can never grant the access, therefore, the mid- and post-conditions in the entry will never be evaluated.

We next present an example of an EACL that governs access to a host.

Entry 1 specifies that user tom@ORGB.EDU can not login to the host.

Entries 2 and 3 mean that user Joe can shut down the host using either X509 or Kerberos for authentication. If the request succeeds, the user ID must be logged. If the operation fails, the system administrator must be notified by e-mail.

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 user ken@ORGA.EDU can login from the specified IP address range, 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 must be updated. The connection duration time must not exceed 8 hours.

```
# EACL for host malta.isi.edu
# EACL entry 1
neg_access_right test host_login
pre_cond_access_id_USER Kerberos5
tom@ORGB.EDU

# EACL entry 2
pos_access_right test host
shut_down
pre_cond_access_id_USER X509
"/C=US/O=Trusted/OU=orgb.edu/CN=Joe"
rr_cond_audit local on:success/userID
post_cond_notify local
on:failure/admin/userID

# EACL entry 3
pos_access_right test host_shut_down
pre_cond_access_id_USER Kerberos5
joe@ORGB.EDU
rr_cond_audit local on:success/userID
post_cond_notify local
on:failure/admin/userID

# EACL entry 4
pos_access_right test host_check_status
pre_cond_location IP 10.1.1.0-10.1.2.255

# EACL entry 5
pos_access_right test host_login
pre_cond_access_id_USER Kerberos5
ken@ORGA.EDU
pre_cond_location IP 10.1.1.0-10.1.2.255
pre_cond_threshold local
≤3failures/day/failed_log
rr_cond_update_log local
on:failure/failed_log/userID
mid_cond_duration local ≤8hrs
```

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 entries which already have been examined take precedence over new entries.

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. We next 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, directory service or in some other way. The application-specific 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 to the GAA-API 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. For further discussion of the policy composition see [12].

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). 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.

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

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

An EACL may specify conditions of different types, e.g., **access identity**, **location** and **audit**. The GAA-API supports registering condition evaluation functions for different condition types.

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 (taking into account the condition type and defining authority fields). To evaluate conditions in the EACL example given earlier, we might register up to 8 functions [3] with the GAA-API. The GAA-API is structured to support the addition of modules for evaluation of new conditions.

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

1. authorization status S_a.
   Indicates whether the request is authorized (GAA_YES), not authorized (GAA_NO) or uncertain (GAA_MAYBE).

2. mid-condition enforcement status S_m.
   Indicates the evaluation status of the mid-conditions ($GAA\_YES/GAA\_NO/GAA\_MAYBE$).

3. post-condition enforcement status S_p.
   Indicates the evaluation status of the post-conditions ($GAA\_YES/GAA\_NO/GAA\_MAYBE$).

The status values are obtained during the evaluation of conditions in the relevant EACL entries:
GAA_YES - all conditions are met;
GAA_NO - at least one of the conditions fails;
GAA_MAYBE - none of the conditions fails but there is at least one condition that is left unevaluated.

The GAA-API returns GAA_MAYBE if the corresponding condition evaluation function is not registered with the API. In some cases, it is convenient to return some of the conditions unevaluated for further evaluation by the calling application.

4 The GAA/ssh Integration

Secure shell (ssh) is being widely deployed because of its features that ensure secure communications across the net-

[3] Depending on the implementation, we may register either one or two functions to evaluate conditions of the same type but with different defining authority fields, e.g., $pre\_cond\_access\_id\_USERX509$ and $pre\_cond\_access\_id\_USERKerberos5$.

work as well as its ease of use. However, correctly configuring the server (sshd) with desired policies is not an easy task, because the authorization policies are described in the server configuration file that contains not only the policies but also the configuration parameters for the server. Hosting two separate functionalities into one configuration file leads to the problem of having inflexible mechanism of describing authorization policies. In addition, the server has to be restarted after modifying the content of the configuration file to reflect changes. If the modification was done to change the policy, instead of the the configuration of the server, restarting the server would prohibit the dynamic response of the server to the potential or actual network threat conditions.

4.1 The Policy Enforcement Process

The GAA-API was integrated into Openssh version 2.9p2 (http://www.openssh.org). The integration contributed only about 250 lines of "glue" code. Only two files auth2.c and serverloop.c were modified and one new file gaa-plug.c (containing the GAA-API initialization and access control calls) was added.

The GAA-API is integrated into ssh by modifying the $userauth\_reply$ function in the file auth2.c. The file serverloop.c was modified to support the execution control and post-execution phases. The GAA/ssh integration is shown in Figure 1.

The GAA-API makes use of system-wide and local configuration and policy files. The configuration files list routines and parameters for evaluating conditions specified in the policy files. The system-wide policy applies to all applications in the system. The local policy describes security requirements of sshd.

1. The **initialization phase**.

   When the sshd starts, first the GAA-API is initialized by calling $gaa\_initialize$ and gaa_new_sc that extract and register condition evaluation and policy retrieval routines from the system and local configuration files, fetch the system policy file, and generate internal structures for later use.

2. The **access control phase** starts with receiving a connection request.

   (a) The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policies associated with the target host. The function reads the system-wide policy file, converts it to the internal EACL representation and places it at the beginning of the list of EACLs. Next, the function retrieves and translates the local policy file and adds it to the list.

   (b) The request is converted into a list of requested access rights. The authenticated user identity is extracted from the $authctxt$ structure and is placed in the gaa_sc security context structure.

   (c) Next, the $gaa\_check\_authorization$ function is called to check whether the requested right is authorized by the ordered list of EACLs. This function finds the EACL entries where the the requested right appears and calls the registered routines to evaluate pre- and request-result conditions in the entries. If there are no pre-conditions, the authorization status is set to GAA_YES. 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 entry, the conditions are evaluated and the intermediate result is calculated. The conjunction of the intermediate result and S_a is stored in the authorization status S_a.

   Based on the authorization status S_a the connection is permitted or rejected as follows:
   S_a = GAA_YES connection is allowed.
   S_a = GAA_NO connection is rejected.
   S_a = GAA_MAYBE connection is rejected.
   The detailed information is returned that lists all matching policy rights and associated conditions, with flags set to indicate whether each condition was evaluated and/or met.

3. The **execution control phase** consists of starting the connection and calling the $gaa\_execution\_control$ function. This function checks whether the mid-conditions associated with the granted access rights are met. If mid-conditions are found, the conditions are evaluated. Some mid-conditions are evaluated just once [4], 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 to enforce the post-conditions associated with the granted rights. This function performs policy enforcement after the operation completes by executing actions such as notifying by email, modifying system variables, writing log file, etc.

   The connection status (indicating whether the connection succeeded/failed) is passed to the

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

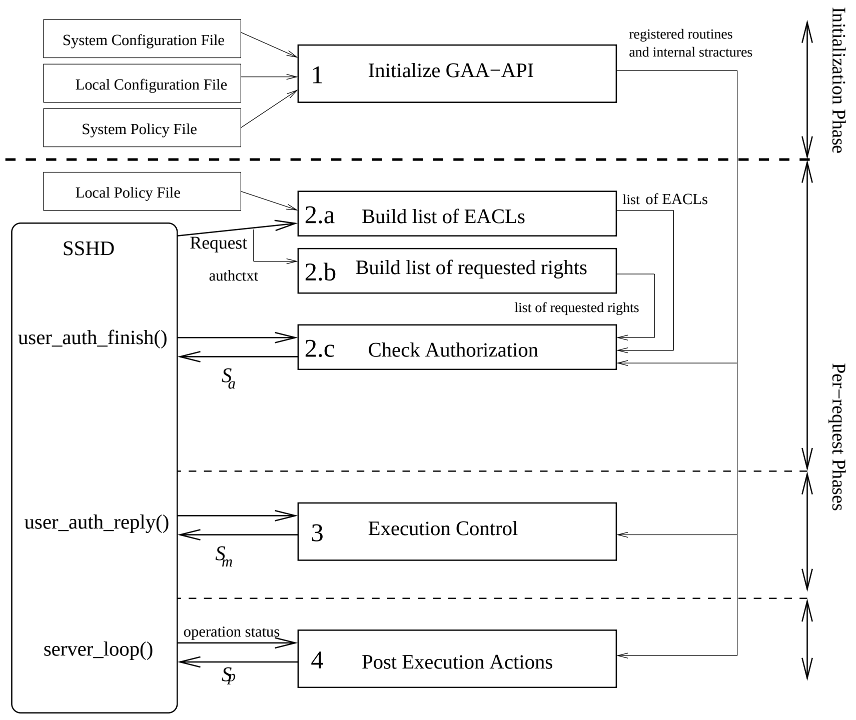

Figure 1. GAA/ssh integration.

$gaa\_post\_execution\_actions$ function. If no post-conditions are found, GAA_YES is returned, otherwise the post-conditions are evaluated and the result is returned in S_p.

5 Deployments

In this section we illustrate how our framework can be deployed to enable fine-grained response to attacks.

5.1 Network Lockdown

This scenario demonstrates how our system adapts the applied authentication policies to require more information from a user when potentially dangerous activity has been detected.

This scenario is designed for organizations with the following characteristics:

- Mixed access to web services. Access to some web resources require user authentication, some do not. 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”.
- Authenticated ssh connections from the Internet are allowed to access hosts on the organization’s LAN.
- A network-based IDS supplies a system threat level. For example, *low* threat level means normal system operational state, *medium* threat level indicates suspicious behavior and *high* threat level means that the system is under attack.
- Policy: **When system threat level is higher than *low*, one needs to lock down the system and require user authentication for all accesses within the LAN.** Strong authentication protects against outside intruders. To some extent, authentication may help to reduce insider misuse. In particular, insiders are discouraged if the identity of a user can be established reliably.

The policy requirements can be represented by the following EACL that protects all ssh and web server connections within the LAN:

```
# EACL entry 1
pos_access_righ apache *
pre_cond_system_threat_level local >low
pre_cond_access_id_USER apache *

# EACL entry 2
pos_access_righ ssh *
pre_cond_system_threat_level local >low
pre_cond_access_id_USER ssh *
```

The pre-conditions in EACL entries 1 and 2 mean that all Apache and ssh accesses have to be authenticated if the system threat level is higher than *low*. Currently the implementation of the `pre_cond_system_threat_level` condition retrievs system threat level from a specific file.

5.2 Application Level Intrusion Detection

We next demonstrate how our framework provides real time application level intrusion/misuse detection capabilities. This example demonstrates detection and response to a particular DoS attack: opening a large number of simultaneous connections to the ssh server starves the number of available sockets, disallowing new connects.

Assume that EACLs that govern hosts within the LAN contain the following EACL entry:

```
pos_access_right ssh host_login
pre_cond_access_id_USER X509 *
pre_cond_threshold local ≤20/user_sessions
rr_cond_notify local
on:failure/admin/ssh,DoS
rr_cond_update_log local
on:failure/failed_log/userID
mid_cond_update_log local
user_sessions/userID+1
post_cond_update_log local
on:success/user_sessions/userID-1
```

Evaluation of the `pre_cond_access_id_USER` asserts a proper user authentication. The pre-condition `pre_cond_threshold` reads the log of active sessions to determine the number of sessions with the user ID field equal to the one in the user ID credentials.

If the number is greater than 20, the request is rejected. The `rr_cond_notify` condition sends e-mail to the system administrator reporting time, user name and a threat type. Next, the `rr_cond_update_log` updates the log of failed logins to include a new suspicious user ID. If the number of such sessions is less than 20, the request is granted, the connection is established and the mid-condition `mid_cond_update_log` is evaluated. This condition is evaluated just once, it updates the number of active ssh connections for the user. After a connection is closed, the post-condition `post_cond_update_log` updates the number of connections reducing it by 1.

6 Performance

7 Related Work

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 [5] 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.

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. 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 [8], [9] 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. [6] 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 [7]. Sandhu and Samarati [13] discuss authentication, access control and intrusion detection technologies and suggest that combination of the techniques is necessary in order to build a secure system.

8 Conclusions and Future Work

Traditional authorization mechanisms check whether a user is acting within prescribed parameters and will not detect abuse of privileges. In this paper we presented an authorization framework that enables the specification and enforcement of workable security policies that govern access to protected resources, identify threats that may occur within application and specify intrusion response actions. The GAA-API combines policy enforcement with application level intrusion detection and response, allowing countermeasures to be applied to ongoing attacks before they cause damage. The GAA-API implementation is available at **http://www.isi.edu/gost/info/gaaapi/source**.
The GAA-API has been integrated with several applications, including Apache web server [12], sshd and FreeS/WAN IPsec for Linux.

Currently, the GAA-API integrated sshd obtains part of the policy from the original sshd configuration file (to maintain the backward compatibility) and uses the policy file specifed in EACL format to supplement the existing policy. We plan to improve the GAA/ssh integration to completely take over the authorization phase of sshd.

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 per each time interval and, therefore, avoid the problem of coordination of multiple condition evaluation processes. However, this approach results in inefficient policy evaluation process and leads to systems that cannot scale to large numbers of objects.

The future directions for this research include exploring extensions to the framework to support: concurrent requests; replication of the evaluation mechanism; concurrent evaluation of conditions within the same request; and distributed policy enforcement.

At this point, the issues of spatial and temporal relationships among the policy computations become critical. Policies that govern the same object may have non-trivial interdependencies which must be carefully analyzed and understood.

Another limitation of the current framework is reliance on a policy administrator for defining condition evaluation order which is then enforeced by the framework. The limited awareness of the spatial and temporal dependencies among security policies may cause inconsistencies and undesirable system behavior. In many cases, administrators may not have a clear picture of the ramifications of policy enforcement actions, therfore enforcing these policies might have unexpected interactive or concurrent behaviour. Automation is essential to minimise human error, and it can only be used safely when there is a formal model that explicitly addresses both the spatial and the temporal aspects of dynamic authorization.

We aim to develop a formal model which can be used to create policies with strong security guarantees, eliminating guesswork in the design and deployment of adaptive security policies.

9 Acknowledgments

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. 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 Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy, the U.S. Government or the Xerox Corporation. Figures and descriptions were provided by the authors and are used with permission.

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] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] T. V. Ryutov, B. C. Neuman, Li Zhou and Dongho Kim.
Integrated Access Control and Intrusion Detection for Web Servers.
In Proceedings of the 23rd International Conference on Distributed Computing Systems, Providence, Rhode Island, May 2003.

[13] R. Sandhu and P. Samarati.
Authentication, Access Control, and Intrusion Detection.

The Computer Science and Engineering Handbook, pages 1929-1948, CRC Press, 1997.