A Flexible and Distributed Architecture to Enforce Dynamic Access Control*

Thierry Sans, Frédéric Cuppens, and Nora Cuppens-Boulahia

GET/ENST Bretagne, 35576 Cesson-Sévigné Cedex, France
{thierry.sans,frederic.cuppens,nora.cuppens}@enst-bretagne.fr

Abstract. Avoiding unauthorized access in an information system usually means enforcing access control mechanisms. Traditional access control only aims at deciding if an access can be granted or not. Dynamic access control goes further as it aims at controlling also if an ongoing access is still authorized while it is running. Rights Expression Languages, such as MPEG-REL, take into account dynamic aspects of access control policy. However, existing access control architectures are not adequate to enforce such dynamic access control. In this paper, we first explain what dynamic access control involves and why existing architectures are not appropriate. We then provide a flexible and distributed architecture where different components interact to enforce dynamic access control. Using temporal logic of actions, we specify the different interactions between components in the architecture and specify more precisely the component in charge of giving the decision. Finally, we discuss about technical and security issues about how the architecture can be implemented to enable Digital Rights Management (DRM) applications.

1 Introduction

Securing an information system usually means avoiding that users have unauthorized access to information. To prevent unauthorized access, we need to specify and enforce access control to regulate who may have access and how to have access to resources managed by the information system. Specifying and deploying an access control process can be divided into three steps: (1) specifying the access control policy, (2) designing a decision mechanism to grant access according to the access control policy and (3) enforcing the decision in an access control application of the information system.

The access control policy aims at specifying who may have access to the information system. This policy is usually represented by Access Control List (ACL). Nowadays, we talk about Rights Expression Languages (REL) [6, 10] as structured and high expressive languages to specify security policies. A REL aims at identifying the entities of the information system (subjects, actions[1], objects[2]) and specifying which subject may perform an action on an object. A subject can be a physical person[3] or an external application interacting with the information system.

* This work was funded by the "ACI Sécurité Informatique: CASC Project".

[1] also called rights or privileges.

[2] also called resources.

[3] A user who interacts with the information system through a GUI.

Please use the following format when citing this chapter:
Author(s) [insert Last name, First-name initial(s)], 2006, in IFIP International Federation for Information Processing, Volume 201, Security and Privacy in Dynamic Environments, eds. Fischer-Hubner, S., Rannenberg, K., Yngstrom, L., Lindskog, S., (Boston: Springer), pp. [insert page numbers].

When a subject wants to perform an access to a resource, the subject makes a request through the frontend application of the information system. From this request and according to the security policy, a decision mechanism allows or not the subject to have an access to the resource. This decision mechanism is based on the underlying access control model [15] implemented by the Rights Expression Language. The decision mechanism must be reliable, deterministic and reproducible [7].

Finally, an architecture must be designed so that given subject can only have an access to a resource through the access control layer. It means that the architecture must be tamper resistant to bypassing the decision and prevent any uncontrolled access to the information.

Existing access control models [15] provide decision mechanisms to reason on the access control policy. These traditional decision mechanisms only aim at enforcing the decision *before* a given action is launched. However, in more recent applications such as Digital Right Management (DRM) [2, 20], we also want to control if this action is still allowed or not while it is rendering. This is because the access decision can change while the action is running. So, the decision mechanism is no longer static but dynamic.

Why do we need ongoing checks when the access is rendering? Because the environment can change and the conditions to allow an access may not be still satisfied then a previous authorization can then be forbidden. In [17], R.Sandhu and J.Park show up that the traditional approach is not sufficient to deal with dynamic requirements. For instance, "a user can use a free Internet access only if an advertisement bar is on". It means that the bar must be active at the beginning of the Internet access and must remain active during the access is running otherwise the access must end. We agree with [17] that traditional access control cannot enforce this kind of requirement. The pre and $ongoing$ models of $UCON_{ABC}$ overcome this limitation. Pre-models are sets of rules which have to be satisfied before the access is granted. On-models (ongoing models) are sets of rules which have to be satisfied while an access is running.

However, Rights Expression Languages and access control models such as $UCON_{ABC}$ that support dynamic access control do not define an appropriate architecture to enforce the underlying decision mechanism. Why do we need a new architecture to enforce dynamic access control? Because existing architecture [11, 4] are not sufficient to enforce ongoing decision mechanism implied by dynamic access control. For instance, if we consider the XACML specification[4] and focus on its enforcement in a AAA Architecture [16, 4], the specification does not take into account such a mechanism. In this architecture, a subject who wants to perform an access, makes a request to a PEP (Policy Enforcement Point), the component in charge of granting an access or not. The PEP then sends a request[5] to a PDP (Policy Decision Point), the component in charge of making the decision. According to the XACML access control policy, the PDP sends back the decision to the PEP. The PEP finally enforces the decision of granting or not the access.

Now, if we want to take into account dynamic access control requirements into the AAA architecture, the only way to perform ongoing checks is to send request to the PDP periodically in order to check if the ongoing access is still allowed or not. This

[4] An OASIS standard in version 2.0.

[5] This request is embedded in a SAML Token. SAML is another OASIS standard.

approach is not adequate for two reasons. First it creates useless traffic between the PEP and PDP. Secondly, it is costly in time to periodically check if the conditions to allow the access are still satisfied or not. It would be more efficient if the PDP can notify the PEP that the access must be revoked.

Thus, the objective of this paper is to formally define an adequate architecture to enforce dynamic access control mechanism. This architecture takes into account dynamic interactions between the component in charge of making a decision and the component in charge of enforcing it. In section 2, we further explain our approach of dynamic access control. In section 3, we present the different components of our architecture and specify how they interact each other. Using temporal logic we define how the components must behave in the architecture in order to enforce dynamic access control. In section 4, we focus on the specification of the component in charge of the decision mechanism. We formally specify how this component makes the decision. Finally in section 5, we discuss how to implement this architecture with existing technologies and also what are the security issues we have to face to reliably deploy this architecture.

2 Our approach

Many Access Control List Languages (ACL) and Rights Expression Languages (REL) has been previously defined [6, 10]. So, this paper does not aim at providing a new structured language to specify the access control policy. We simply assume that the expression of the access control policy is based on provisional authorizations [12]. It means that an access can be allowed if some conditions are satisfied. These conditions may depend on the system environment [15]. They can change while an access is running and require dynamic access control mechanisms to be implemented.

In order to represent provisional authorizations, we adopt the MPEG-REL[6] approach [10]. Even if the initial specification of MPEG-REL enforcement is not formally defined, it has been formalized in [8]. Enforcement of the access control policy is divided into two steps. In the first step, called "policy matching", we have to select the authorizations relevant to the requested access. At this step, the access can be denied if none authorization can match the request. In the second step called "condition validation", the conditions associated with the provisional relevant authorizations are evaluated with respect to the system environment, and the access is either allowed if one condition applies or denied if all conditions fail. In dynamic access control, it is necessary to make a clear separation between policy matching and condition validation. Indeed, while the access is running, conditions continue to apply and we must check if they are still satisfied. By contrast, the policy does not need to be matched again, except when the policy is updated during the access. We deal with this last issue in the policy update extension specified further in section 4.3.

Once the decision mechanism is defined, we aim at designing an appropriate architecture to enforce dynamic access control. For this purpose, we suggest an approach based on a mathematical formalism to model interactions between components used to enforce dynamic access control. We have chosen the Temporal Logic of Actions (TLA)

[6] Called XrML before becoming an ISO/IEC Standard.

defined by L.Lamport [14]. TLA is based on first order logic extended with temporal modalities. We briefly introduce here the main concepts of the TLA formalism :

A state is an assignment of values to variables.

A predicate is a boolean expression valuated from a given state :
s_i: $s_i[\![P]\!] \triangleq P(\forall v : s_i[\![v]\!]/v)$

An action is a boolean expression representing a transition between an old state and a new state of the system. So, actions are expressed with unprimed and primed variables representing respectively a condition on the old system state and a condition on the new system state. An action can be valuated from two consecutive states: $s_i[\![A]\!]s_{i+1} \triangleq A(\forall v : s_i[\![v]\!]/v, s_{i+1}[\![v]\!]/v')$. The meaning $[\![A]\!]$ of an action A is a relation between states, i.e. a function that assigns a boolean $s_i[\![A]\!]s_{i+1}$ to a pair of states s_i and s_{i+1}.

For a given sequence of states $\langle s_0, s_1, s_2, ..., s_n\rangle$, for A and B actions, TLA defines the following modalities:

	$\langle s_0, s_1, s_2, ..., s_n\rangle[\![A]\!]$	$\triangleq s_0[\![A]\!]s_1$
Next $\bigcirc A$	: $\langle s_0, s_1, s_2, ..., s_n\rangle[\![\bigcirc A]\!]$	$\triangleq s_1[\![A]\!]s_2$
Always $\Box A$	: $\langle s_0, s_1, s_2, ..., s_n\rangle[\![\Box A]\!]$	$\triangleq \forall i \in [0..n]\ s_i[\![A]\!]s_{i+1}$
Eventually $\Diamond A$	: $\langle s_0, s_1, s_2, ..., s_n\rangle[\![\Diamond A]\!]$	$\triangleq \exists i \in [0..n]\ s_i[\![A]\!]s_{i+1}$

Then, using first order logic operators, a TLA formula is defined by the following grammar:

Formula F ::= $A \mid \neg F \mid \bigcirc F \mid \Box F \mid \Diamond F \mid F \wedge F \mid F \vee F \mid F \rightarrow F$

3 The architecture

In this section, we introduce the different components of our distributed architecture. Using TLA, we show how they interact and behave to enforce dynamic access control.

3.1 The components

We focus on three main components of our access control architecture: (1) The frontend handler, (2) the rendering handler and (3) the policy handler.

The frontend handler is the application layer where a subject can interact with the information system. Using this component, the subject requests execution of actions managed by the system.

The rendering handler is in charge of performing an allowed request. It executes the corresponding action from the action library. The action library is a set of programs in charge of rendering the information to the subject.

The policy handler is in charge of evaluating a request and provides a decision with respect to the access control policy. This component enforces the decision mechanism of the underlying Right Expression Language used to express the policy. It can then allow a subject to perform an action requested from the frontend handler. This decision

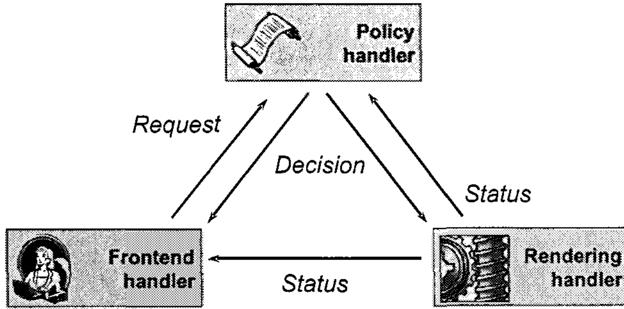

Fig. 1. Distributed Architecture.

message is sent either to the rendering handler or the frontend handler. With dynamic access control, if the decision changes during the rendering, the policy handler notifies the rendering handler that the action is no longer allowed and stops the rendering.

As shown in figure 1, each component interacts with the others by exchanging messages. In order to enforce dynamic access control, we define three message classes:

- Request: When a subject wants to have access to the system, the *request* message is sent from the frontend handler to the policy handler to be evaluated.
- Decision: There are three types of decision messages: *grant*, *deny* and *revoke*. The *grant* message is sent to the rendering handler when a request is allowed. The *deny* message notifies the frontend handler when a request is not allowed. In dynamic access control, we also want to control if an ongoing access is still allowed. If the decision changes during the access, a *revoke* message is sent to the rendering handler to stop the rendering.
- Status: Once a decision is taken, the rendering handler must notify the other handlers that the rendering action is launched or is completed. Thus, we define the *access* message when the action is launched and the *end* message when it is completed.

For a better reading, we have chosen to define data structures with a TLA-like formalism. We formally define these different types of message as follows:

$$\text{TYPE} \qquad MessageType = \{\ \text{“}request\text{”},\ \text{“}grant\text{”},\ \text{“}deny\text{”},\ \text{“}revoke\text{”},\ \text{“}access\text{”},\ \text{“}end\text{”}\ \}$$

Every message contains a description of the access. We call *target* a tuple composed of: (1) who wants to access, (2) which object is requested and (3) for which rendering action. The target is denoted α and defined as follows:

$$\text{TYPE} \qquad Target = [Subject \times Action \times Object]$$

A *message* is then formally defined as a tuple with a message type and a target as follows:

$$\text{TYPE} \qquad Message = [MessageType \times Target]$$

Finally, we need to model interaction between components through the network by sending and receiving messages. The network is a set of messages that we denote Φ. We respectively define two predicates snd and rcv like this:

$$\text{PREDICATE} \quad rcv = [Message \rightarrow Boolean]$$
$$snd = [Message \rightarrow Boolean]$$

These predicates modify the network state by adding or taking off a message. So, invoking one of these predicates will modify Φ. For instance, $\forall\, \omega : Message\ \Phi, \Phi' \vdash rcv(\omega)$ means that a message is taken from Φ and now the set of messages, representing the network, is Φ'.

Notice that for a better reading of formulae, we do not need to explicitly specify the sender and the receiver in the snd and rcv predicates. Using the type of the messages, every component knows who is the sender and the recipient.

In the next part, we further explain how the different components interact using messages, and formalize how they enforce dynamic access control policy.

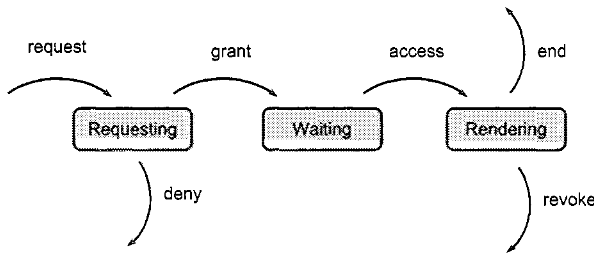

Fig. 2. Dynamic Access Control Enforcement.

3.2 The dynamic access control enforcement

Figure 1 shows what kind of message the different components exchange and figure 2 shows when the different messages are sent.

First, a request message is sent from the frontend layer to the policy handler. This *request* message contains the target representing who wants to access, which object is requested and for which rendering action. Then, the policy handler makes a decision with respect to the access control policy. If the decision is to allow the request, a *grant* message is sent to the rendering handler. If the decision is to not allow the request, a *deny* message is sent to the frontend handler notifying the subject that the request is rejected. So, using temporal logic, we can describe the behavior of the policy handler. When a *request* message is sent, the policy handler has to send a *grant* message or a *deny* message in return. We can formalize it as follows:

The policy handler :

$$\Box\, [\, rcv(\text{“}request\text{”}, \alpha) \;\rightarrow\; \Diamond\, (\, snd(\text{“}grant\text{”}, \alpha) \;\vee\; snd(\text{“}deny\text{”}, \alpha)\,)\,]$$

Notice that the formula, used to represent how the policy handler behaves, is indeterministic. The choice between *grant* and *deny* message depends on the decision

mechanism implemented by the policy handler. This non determinism will disappear when we shall formally specify the decision mechanism of the policy handler in section 4.

So far, we have described the traditional access control enforcement in our distributed architecture. Now, we want to enforce dynamic access control and check if an ongoing rendering is still allowed or not. When the rendering handler receives a *grant* message, it must perform the corresponding action from the actions library to satisfy the authorized request. The rendering handler then sends an *access* message to the policy handler notifying that the authorized access has been launched. So, when a request is allowed by the policy handler, the action will finally be launched by the rendering handler. For the rendering handler, it means that when a *grant* message is received, it has to send an *access* message in response. Respectively, for the policy handler, when a message *grant* is sent, it expects to receive back an *access* message. We can formalized it as follows:

The rendering handler :
$\Box\,[\, rcv(\text{"}grant\text{"}, \alpha) \;\rightarrow\; \Diamond\,(snd(\text{"}access\text{"}, \alpha)\,)\,]$
The policy handler :
$\Box\,[\, snd(\text{"}grant\text{"}, \alpha) \;\rightarrow\; \Diamond\,(rcv(\text{"}access\text{"}, \alpha)\,)\,]$

With the dynamic approach, while the action is rendering, the policy handler is still evaluating the request until it ends. If the decision changes during the rendering, the policy handler notifies the rendering handler that the action is no longer allowed and stops the rendering. If the decision does not change, the action will eventually end and the rendering handler will send an *end* message to notify the policy handler that it has to stop evaluating the decision for the corresponding request. So, when an action is rendering, the policy handler can revoke the access with a *revoke* message. If the decision does not change, the policy handler will eventually receive the notification with an *end* message that the action has been done. Respectively, we can formalize the corresponding behavior for the rendering handler. The different behaviors can be formalized as follows:

The policy handler :
$\Box\,[\, rcv(\text{"}access\text{"}, \alpha) \;\rightarrow\; \Diamond\,(\, snd(\text{"}revoke\text{"}, \alpha) \,\vee\, rcv(\text{"}end\text{"}, \alpha)\,)\,]$
$\Rightarrow\; \Box\,[\, rcv(\text{"}request\text{"}, \alpha) \;\rightarrow\; \Diamond\,(\, snd(\text{"}deny\text{"}, \alpha)$
$\qquad\qquad \vee\, snd(\text{"}revoke\text{"}, \alpha) \,\vee\, rcv(\text{"}end\text{"}, \alpha)\,)\,]$
The rendering handler :
$\Box\,[\, snd(\text{"}access\text{"}, \alpha) \;\rightarrow\; \Diamond\,(\, snd(\text{"}end\text{"}, \alpha) \,\vee\, rcv(\text{"}revoke\text{"}, \alpha)\,)\,]$
$\Rightarrow\; \Box\,[\, rcv(\text{"}grant\text{"}, \alpha) \;\rightarrow\; \Diamond\,(\, snd(\text{"}end\text{"}, \alpha) \,\vee\, rcv(\text{"}revoke\text{"}, \alpha)\,)\,]$

Finally, we can give the behavior of the frontend handler. When it has sent a request, the frontend handler expects to receive either: A *deny* message notifying that the corresponding request has not been allowed. A *revoke* message notifying that the corresponding request has been allowed and launched, but it has not been allowed to complete. An *end* message notifying that the corresponding request has been allowed and has successfully completed.

So, the frontend handler behavior can be formalized like this:

The frontend handler :

$$\Rightarrow \Box\,[\, snd(\text{“}request\text{”}, \alpha) \;\rightarrow\; \Diamond\,(\, rcv(\text{“}deny\text{”}, \alpha) \;\vee\; rcv(\text{“}revoke\text{”}, \alpha) \;\vee\; rcv(\text{“}end\text{”}, \alpha)\,)\,]$$

4 The policy handler specification

In this section, we focus on the specification of the component that enforces the access control policy: The policy handler. First, we model how to manage provisional conditions to make a decision. Then, using TLA, we formally specify how the policy handler enforces this decision.

4.1 Managing provisional conditions

We denote Γ the access control policy. According to the previous definition of $Target$ given in section 3.1, we define how to match the policy and the target:

PREDICATE $\quad isPermitted = [Target \times Condition \rightarrow Boolean]$

For instance, $\forall\, \alpha : Target,\ \forall\, c : Condition\ \Gamma \vdash isPermitted(\alpha, c)$ means that, according to the security policy Γ it is allowed to grant an access α if the condition c is satisfied. Notice that this predicate does not change the access control policy.

We then both check if a condition is satisfied in the environment and also log a message as environment information:

PREDICATE $\quad isSatisfied = [Target \times Condition \rightarrow Boolean]$
$\qquad\qquad\quad addLog = [Message \rightarrow Boolean]$

For instance, $\forall\, \alpha : Target,\ \forall\, c : Condition\ \Sigma \vdash isSatisfied(\alpha, c)$ means that, according to the environment Σ, the condition c that belongs to α is satisfied. In the same way, $\forall\, \omega : Message\ \Sigma, \Sigma' \vdash addLog(\omega)$ modifies the environment by logging a message ω. Notice that the predicate $isSatified$ does not change the environment whereas $addLog$ modifies it.

4.2 Policy Handler TLA Specification

As shown in figure 3, there are three steps in the decision enforcement: (1) getting a message from the network and log it, (2) making a decision according to the message type and (3) checking ongoing access. The policy handler makes a decision as follows:

- Request: If the policy matches the target and if the corresponding conditions are satisfied then a *grant* message is sent to the rendering handler, else a *deny* message is sent to notify the application handler. This case is defined in the *requesting* action.
- Access: If the policy matches the target then a corresponding $\langle target, condition\rangle$ is added in the *ongoing* set, else a *revoke* message is sent to the rendering handler. This case is defined in the *accessing* action.

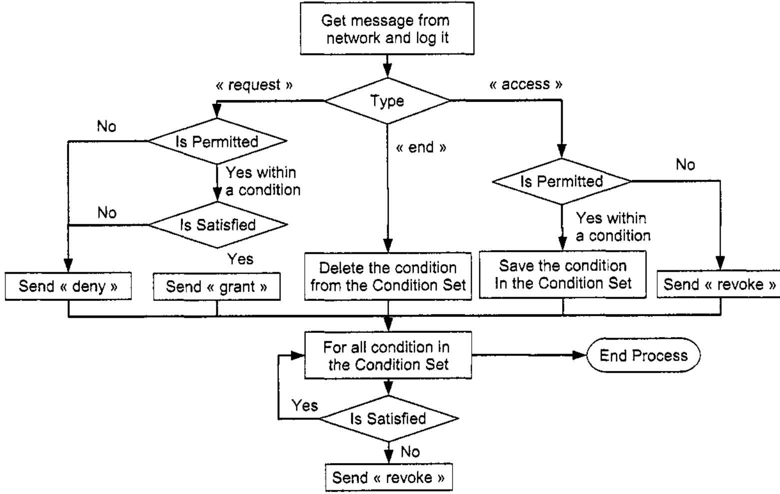

Fig. 3. Policy Handler Algorithm.

- End : The corresponding target is deleted from the *ongoing* set to stop validation of ongoing access. This case is defined in the *ending* action.

In the specification, we need a variable *ongoing* to represent the set of ongoing access and the corresponding condition to be satisfied:

VARIABLE $\quad ongoing : \{[Target \times Condition]\}$

$$
\begin{aligned}
Requesting \triangleq \quad & \exists\, \alpha : Target,\ \exists\, c : Condition \mid \\
& \Phi, \Phi' \vdash rcv(\langle\text{“}request\text{”}, \alpha\rangle) \\
& \wedge\, \Sigma, \Sigma' \vdash addLog(\langle\text{“}request\text{”}, \alpha\rangle) \\
& \wedge\, If\ (\ \Gamma \vdash isPermitted(\alpha, c) \wedge \Sigma \vdash isSatisfied(\alpha, c)) \\
& \qquad Then\ \Phi, \Phi' \vdash snd(\langle\text{“}grant\text{”}, \alpha\rangle) \\
& \qquad Else\ \Phi, \Phi' \vdash snd(\langle\text{“}deny\text{”}, \alpha\rangle)
\end{aligned}
$$

$$
\begin{aligned}
Accessing \triangleq \quad & \exists\, \alpha : Target,\ \exists\, c : Condition \mid \\
& \Phi, \Phi' \vdash rcv(\langle\text{“}access\text{”}, \alpha\rangle) \\
& \wedge\, \Sigma, \Sigma' \vdash addLog(\langle\text{“}access\text{”}, \alpha\rangle) \\
& \wedge\, If\ (\ \Gamma \vdash isPermitted(\alpha, c)) \\
& \qquad Then\ ongoing' = ongoing \cup \{\langle\alpha, c\rangle\} \\
& \qquad Else\ \Phi, \Phi' \vdash snd(\langle\text{“}revoke\text{”}, \alpha\rangle)
\end{aligned}
$$

$$
\begin{aligned}
Ending \triangleq \quad & \exists\, \alpha : Target,\ \forall\, c : Condition \mid \\
& \Phi, \Phi' \vdash rcv(\langle\text{“}end\text{”}, \alpha\rangle) \\
& \wedge\, \Sigma, \Sigma' \vdash addLog(\langle\text{“}end\text{”}, \alpha\rangle) \\
& \wedge\, ongoing' = ongoing \setminus \{\langle\alpha, c\rangle\}
\end{aligned}
$$

Finally, the policy handler checks if all ongoing actions are still allowed. So, for each target in the *ongoingset*, if the condition is satisfied then the target is kept in the set, else a *revoke* message is sent and the message is deleted from the *ongoing* set. This is defined in the *checking* action.

$$\begin{array}{lll} Checking & \triangleq & \forall\, \alpha : Target,\ \forall\, c : Condition\ | \\ & & \langle \alpha, c \rangle\ \in ongoing \\ & & \wedge\, If\ (\ \Sigma \vdash isSatified(\alpha, c)) \\ & & \quad Then\ \langle \alpha, c \rangle\ \in ongoing' \\ & & \quad Else\ \Phi, \Phi' \vdash snd(\langle \text{“}revoke\text{”}, \alpha \rangle) \end{array}$$

To complete the specification, we then define the policy handler specification as follows:

$$\begin{array}{lll} Init & \triangleq & ongoing = \emptyset \\ PolicyHandler & \triangleq & Init\ \wedge \Box [\ \ Requesting\ \vee Accessing \\ & & \qquad\qquad \vee\ Ending\ \ \vee Checking\ ]_{<ongoing>} \end{array}$$

4.3 The update extension

What happens if the policy changes during an authorized access? This can change the conditions associated with ongoing access. So, we provide an update extension to deal with this problem. We can assume that the policy is managed by a new component: a policy manager. The policy manager can be seen as a rendering handler that enforces updates of the security policy. Therefore, an update must have been previously allowed by an administrative policy. This kind of reflexive architecture can enforce a reflexive access control administration model like [15].

When an update is done, the policy manager (like a rendering handler) sends to the policy handler an end message $\langle \text{“}end\text{”}, \langle s, \text{“}modify\text{”}, \text{“}securityPolicy\text{”} \rangle \rangle$ notifying that the policy has changed. We then need to extend the definition of the “ending” action in order to handle an update. So, we modify the *ending* action as follows:

$$\begin{array}{ll} Ending\ \ \triangleq & \exists\, \alpha : Target,\ \forall\, c : Condition\ | \\ & \Phi, \Phi' \vdash rcv(\langle \text{“}end\text{”}, \alpha \rangle) \\ & \wedge\, \Sigma, \Sigma' \vdash addLog(\langle \text{“}end\text{”}, \alpha \rangle) \\ & \wedge\, ongoing' = ongoing \backslash \{ \langle \alpha, c \rangle \} \\ & \wedge\, If\ (\exists\, s : Subject\ |\ \alpha =\ \langle s, \text{“}modify\text{”}, \text{“}SecurityPolicy\text{”} \rangle) \\ & \quad Then\ Updating_{<ongoing>} \end{array}$$

When an update happens, the *updating* action specifies that for all ongoing access, the policy handler checks the policy again and updates the ongoing set with the corresponding condition. Some access can be revoked if the policy does not authorized them anymore. The *updating* action is defined as follows:

$$\begin{array}{lll} Updating & \triangleq & \forall\, \alpha : Target,\ \forall\, c_1, c_2 : Condition\ | \\ & & \wedge\, \langle \alpha, c_1 \rangle\ \in ongoing \\ & & \wedge\, If\ (\ \Gamma \vdash isPermitted(\alpha, c_2)) \\ & & \quad Then\ \langle \alpha, c_2 \rangle\ \in ongoing' \\ & & \quad Else\ \Phi, \Phi' \vdash snd(\langle \text{”}revoke\text{”}, \alpha \rangle) \end{array}$$

Notice that at the *updating* step, the conditions do not need to be validated because they are going to be validated in the next step with the *checking* action as defined previously in the policy handler specification.

5 Technical and security issues

In traditional access control architecture, the data and rendering actions remain on the server side in a secure area. Identification, authentication and access control mechanisms enable a trusted subject to interact with the information system from a (potentially) non secure client. However, information systems are currently getting more complex and distributed. In this context [2, 20], the data and rendering actions are more and more disseminated on the client side. So, we need a flexible and distributed architecture to control the outsourced information.

To deal with new communication models like DRM, our proposal can be implemented as a service oriented architecture (SOA) as defined in [5]. Components of our architecture are independent from each other and can be implemented both on a server side or client side. The implementation of the communication layer can be based on web services [1, 19]. Targets of access control could be wrapped in SAML messages as described in [9]. This architecture enforces security mechanisms, so it must be tamper resistant to different attacks. Let us focus on security issues of this architecture:

Trust management: Only trusted components can interact in the architecture [3], because a rogue policy handler might deliver malicious decision to perform an unauthorized access. To prevent this flaw, authentication mechanism must be enforced between the different components of the trusted architecture.

Integrity: The messages must not be modified by a non authorized entity. Digital signatures must be applied to messages exchanged between trusted components to guarantee the integrity of the access control information. Messages must also not be deleted without control, so non repudiation mechanism [21] must be enforced to guarantee that a message sent will be eventually received.

Confidentiality: At first glance, we can think that confidentiality is not necessary in our architecture. It could not matter if someone can see access control decisions and status. However, confidentiality can be required when the target embeds private or confidential information about subject, action and resource [13].

Availability: For instance, it could be obvious to let the policy handler allows an access and then, performs a denial of service on this component to avoid a *revoke* message. To prevent this intrusion, heartbeats messages must be exchanged between the components during an access.

Now, another security issue can appear regarding the component location in the information system.

On a server-side, we consider that the servers[7] are inside the trusted and secure area. It means that we assume that the underlying Operating System (OS) and program execution is safe from any external threat.

[7] When different servers host the different components of our architecture.

On a client-side, we can no longer assume this hypothesis. The component is executed out of the scope of the safe and secure area. On the client side, the running OS and processes execution can be hooked. Existing DRM frameworks usually deal with this issue. Most of the time the application handler and rendering handler are embedded in a program executed in the client side. An obvious solution could be to hide the code of both the OS and the program to avoid modification. Unfortunately this solution is not viable because it cannot be enforced in open source systems, and also because reverse engineering is still possible. The solution could be to use a Trusted Third Party (TTP) in the client. This TTP is a safe area where trusted programs can be executed. New technology like TCPA[8] [18, 2] embeds this TTP as a part of the processor chip.

6 Conclusion

In traditional access control architecture, access control only applies before launching an action. Dynamic access control goes one step further by also controlling if provisional conditions are still satisfied during the access. The main contribution of this paper is to formally define a distributed architecture that supports dynamic access control mechanisms.

Our architecture is based on three components: (1) The request handler is used by an external subject to make a request to the information system, (2) the policy handler is in charge of making a decision and (3) the rendering handler is in charge of enforcing the decision associated with a given allowed request. We first show that existing architectures are not relevant to enforce dynamic access control because interactions between their policy handler and their rendering handler are not efficient.

Our architecture defines new interactions between the policy handler and the rendering handler. When the access is granted, the rendering policy launches the action and notifies it to the policy handler. During the access, the policy handler keeps on checking if the conditions allowing the access are still satisfied. If they are not, the policy handler notifies the rendering handler to revoke the access. Else, if the action ends normally, the rendering handler notifies it to the policy handler in order to stop ongoing checks.

We then formally specify how the policy handler makes a decision using temporal logic of actions. First, we define what the policy handler does when it receives access control messages. Secondly, we define an endless action to check conditions that belongs to ongoing access.

Our architecture can be implemented as a service oriented architecture (SOA). Components can be either distributed on a server side or on a client side. Therefore, our architecture can be used as a framework to enforce Digital Rights Management (DRM) applications.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[8] Trusted Computing Platform Alliance.

2. Eberhard Becker, Willms Buhse, Dirk Gnnewig, and Niels Rump, editors. *Digital Rights Management: Technological, Economic, Legal and Political Aspects*. Springer-Verlag GmbH, 2003.
3. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *The IEEE Symposium on Security and Privacy (SSP'96)*, Oakland, CA, May 1996.
4. C.T.A.M de Laat, G.M. Gross, L.Gomans, J.R. Vollbrecht, and D.W. Spence. Generic AAA architecture, RFC 2903. Technical report, IETF Internet Working Group, August 2000.
5. Thomas Erl. *Service-oriented Architecture: Concepts, Technology, And Design*. Prentice Hall PTR, 2005.
6. Susanne Guth. *Digital Rights Management: Technological, Economic, Legal and Political Aspects*, volume Volume 2770, chapter 2.3.5 Rights Expression Languages, pages 101–112. Springer-Verlag GmbH, 2003.
7. J.Y. Halpern and V. Weissman. Using First-Order Logic to Reason about policies. In *16th IEEE Computer Security Foundation Workshop (CSFW'03)*, Pacific Grove, California, June 2003.
8. J.Y. Halpern and V. Weissman. A Formal Foundation for XrML. In *17th IEEE Computer Security Foundation Workshop (CSFW'04)*, Pacific Grove, California, June 2004.
9. J. Hughes and E. Maler. Security Assertion Markup Language (SAML) Version 2.0. Technical report, OASIS, February 2005. www.oasis-open.org.
10. International Organization for Standardization (ISO). *ISO/IEC 21000-5:2004 Information technology – Multimedia framework (MPEG-21) – Part 5: Rights Expression Language*, 2004. www.iso.ch/iso/fr/prods-services/popstds/mpeg.html.
11. ISO International Organization for Standardization. Information technology – Open Systems Interconnection – Security Frameworks in Open Systems – Part 3:Access Control, ISO/IEC 10181-3. Technical report, ISO JTC 1, August 2001.
12. S. Jajodia, M. Kudo, and V.S. Subrahmanian. Provisional Authorizations. In A. Ghosh, editor, *Security and Privacy in E-Commerce*, pages 133–159, Athens, Greece,, November 2000. Kluwer Academic Publishers.
13. L. Korba and S. Kenny. Applying Digital Rights Management Systems to Privacy Right Management. *Journal of Computer and Security*, 2002.
14. Leslie Lamport. *Specifying Systems*. Addison-Wesley Professional, July 2002.
15. Alexandre Miège. *Definition of a formal framework for specifying security policies. The Or-BAC model and extensions*. PhD thesis, ENST, 2005.
16. Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. Technical report, OASIS, February 2005. www.oasis-open.org.
17. J. Park and R. Sandhu. The $UCON_{ABC}$ Usage Control Model. *ACM Transactions on Information and System Security*, 7(1), February 2004.
18. Siani Pearson. *Trusted Computing Platforms: TCPA Technology In Context*. Prentice Hall PTR, July 2002.
19. J. Rosenberg and D. Remy. *Securing Web Services With Ws-Security*. Sams, 2004.
20. W. Rosenblatt, B. Rosenblatt, and W. Trippe. *Digital Rights Management: Business and Technology*. Wiley, Decembre 2001.
21. J. Zhou and D. Gollman. A fair non-repudiation protocol. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 55, Washington, DC, USA, 1996. IEEE Computer Society.