Software Engineering for Distributed systems Security: A practical approach based on problem-oriented language paradigm

Hedi Hamdi[1,2]
1.Jouf University Main Campus, College of Computer and Information Sciences, Sakakah, KSA
2. University Of Manouba, University Campus of Manouba, 2010, Tunisia
hhamdi@ju.edu.sa

Abstract

Securing distributed systems remains a significant challenge for several reasons. First, the security features required in an application may depend on the environment in which the application is operating, the type of data exchanged, and the capability of the end-points of communication. Second, the security mechanisms deployed could apply to both communication and application layers in the system, making it difficult to understand and manage overall system security. This paper presents a policy-based approach to meeting these needs. We propose a framework based on a Domain-Specific Language for the specification, verification and implementation of security policies for distributed systems. Based on a set of abstractions, this framework allows to develop modular security policies and independent of the underlying system. Thus, security policies can be developed by a developer who is not necessarily computer security expert.

Keywords: *DSL, Security policy, compilation, specification, verification, implementation*

1. Introduction

Distributed systems security is definitely not a new topic, since its history starts in the eighties [10]. However, only recently, more emphasis was placed on distributed systems security, and is considered today one of the main issues in the research field of distributed systems. This situation is the result of two main factors:

- The wide usage distributed computing architecture by institutions, companies and individuals. In fact, a variety of distributed computing applications have grown up in recent years. Examples include conferencing applications, messaging applications, peer-to-peer file sharing systems.
- On the one hand, many of these applications are currently used without any security, partly because incorporating security in them is difficult and remains a significant challenge for both developers and as well end-users of the system. On the other hand, the increasing number distributed systems security criminals such as hackers and attackers.

Some problems in incorporating security include:

- Security Requirements are heterogeneous: Security measures are not appropriate for all instances of an application. For example, in a resource distribution application, the security measure required is likely to depend on the nature of the resource (e.g., public or confidential), the operating environment (e.g., private network or public network), the capabilities and policies of the users, etc. Configuring the security incorrectly does have a potential cost: the system may become unavailable to some users, it may become more difficult to use or access, or it may perform poorly.
- Incorporating Security Requirements is difficult: Even after the appropriate of security measure can be determined for an application instance, incorporating it into an application can be non-trivial for developers.

- Layering Problem: It is usually not sufficient to apply security at one layer in the system. Thus, the security mechanisms deployed will often apply to multiple layers in the system, making it difficult to understand and manage overall system security.

Research on security of distributed systems has been pointed mainly on the definition of security protocols, security mechanisms and other techniques solutions. Yet, it has been amply demonstrated in recent years that distributed security is not simply a technical matter, and security solutions cannot be blindly inserted into the systems, but should support both self-adapting and the dynamically change on system behavior [11]. Policy-based security has emerged as a promising solution to address distributed systems security problems. In such approach, policy can be configured to meet the different security requirements with less changes to the application. Policy has been used in different contexts, such as authorization and access control [3],[2], peer session security [3], quality of service guarantees and network configuration. However, the challenge in this approach is to find a suitable framework for policy specification, verification and implementation. In fact, work on the description and implementation of policies is still rare. Following this argument, a software engineering paradigm, namely problem-oriented languages (also called Domain-specific (DSLs), application-oriented, special purpose, task-specific, problem-oriented, specialized, or application languages), have been proposed as promising paradigms for securing distributed systems. Building on our previous work [20],[21] in this paper we propose a new approach for specification, verification and implementation of security policies. Our approach is to apply problem-oriented languages technology to security policies to solve problems listed in this domain. Based on a thorough domain analysis, we have designed our framework, which consists of a support for verification and implementation of policies and a domain specific language.

The remainder of this paper is organized as follows. The following section presents the policy framework requirements. Section 3 discusses features expected of a policy framework. Section 4 describes the design of our framework. Section 5 introduces the PPL Policy Programming language, focusing on the main language abstractions. Section 7 presents an algorithmic solution to policy analysis. Section 8 discusses the related work, and Section 8 concludes the paper and discusses future work.

2. Requirements for a Policy Framework

In this section, we circumscribe the requirements to be considered when defining a framework for security policies. The essential requirement is the design of a language to specify security policies. In addition, the framework should also address the issue of policy implementation for a variety of applications and across different platforms and systems..

2.1. Desirable characteristics of a security policy language

The following is a list of desirable characteristics of a useful security policy language:

Clear statement of policy A policy stated in the language should be clear and be obvious in its meaning. There should be no ambiguity in the meaning of the policy.

Executable The language should be such that it lends itself to being enforced on a system. A stated policy should be able to be generated into implementation mechanisms. This would allow the expressed security properties to be maintained.

Portable The language should be able to express policies for a variety of systems, both common and custom. This would avoid needing to learn a new language for each system with which one is concerned. In addition, when a new system is created, one would not need to invent a new policy language.

Expressive The policy language should be able to express the policy the creator wishes to enforce. There should be a minimum of needs to compromise on intention or to unnecessarily represent it using multiple expressions of policy, as these are failures to express what the policy creator wants. Current models of security should expressible in the language.

Declarative The language should be able to express the policy at the level of detail desired for an application. This would allow policies to be stated at multiple levels of abstraction. Thus, the policy can convey the security intent of the creator. In addition, this ability would be useful in a framework for translating high level policies into lower level policies and enforcement mechanisms –the same language can be used to represent requirements throughout.

Simple Policies expressed in the language should be easy to write, modify, and understand for both casual and experienced users. This will increase the likelihood that the language will be used and that it will embody the policy the user intended.

Allow policies validation and verification the language should lend itself to reasoning about the completeness and correctness of policies stated in it and to other applications of formal reasoning. Thus, one can be sure of the properties derived for policies stated in the language.

2.1. Tool for the deployment of policy

Policy enforcement techniques evolve into highly distributed paradigms, using technologies such as distributed objects, mobile code, intelligent multi-agent systems or programmable networks. We identify below requirements for a policies deployment and implementation tool:

– Support for compiling policies in different execution representations based on the properties and capabilities of the underlying services or applications.
– Flexible enforcement of distributed access control policies on different security platforms.
– Automation of policy distribution to their implementation components.
– Support for existing access control mechanisms.
– Provide the necessary tools to administer policies for large scale systems. These tools should hide implementation level details, allow for the specification, deployment, and coordination of policies within the system, and make it easier to review policies per user as well as per device.

3. Features expected of a policy framework

Our main objective is to design a development framework to facilitate the specification, the verification and the implementation of security policies. This framework must be designed as an architecture that allows easy specification of security policies, and provides the means for detecting conflicts in these policies and for automatically generate the necessary mechanisms for their implementation.

This framework is intended to be flexible, scalable and compatible with existing infrastructures. Based on a thorough domain analysis, we found that such a development framework must provide the following features.

Bringing and facilitate the implementation of the specification to ensure that specification and implementation of a policy is the simplest and the most natural as possible. It must be easy to develop new security solutions or extend existing solutions by reusing existing policies. The specific infrastructure shall be covered up and the code dependent on this infrastructure should be minimized or automatically generated.

Robustness Because a security policy is a critical component in security systems, security policies must be deployed carefully checked. The process of detection and resolution of conflicts in a policy must be treated as a very important element of security policy system. Indeed, the deployment of conflicting policies in a system, creates unexpected behaviour and cause unexpected damage. Therefore, deploy conflicting security policies in a system is probably more devastating than leave the system without security measures.

Performance Given the impact of security measures on the overall performance of a given system and applications, security policies deployed should not introduce significant loss of performance and must ensure the effectiveness of the system in which they are deployed.

4. A framework for specification, verification and implementation of security policies

4.1 Overview

Our framework for specification, verification and implementation of security policies, called PPL **P***olicy* **P***rogramming* **L***anguage*, revolves around two main axes:

– An extensible deployment support for integration and implementation of security policies and
– A DSL allowing easy specification of security policies with the idiom level of abstraction in the security domain. Consequently, domain experts can use it to understand, to validate, to modify, and often to develop security policies.

Deployment support allows to verify and to implement security policies in the target system. The domain specific language provides the specific abstractions to allow the specification of security policies facilitating their development and their integration into deployment support. Figure 1 describes the development process and the components of the PPL architecture.

First, a security policy is specified using the abstractions provided by the PPL language. After syntactic analysis, this policy is subject to a checker to detect potential conflicts, and certifies its correction compared the properties specific to the application or to domain to secure. Finally, the policy is translated into the appropriate representations of security mechanisms of underlying system.

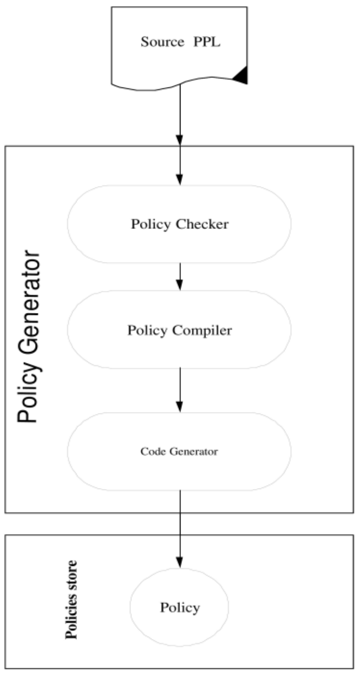

Figure 1: PPL development process

4.2 Architecture

PPL language has not only made a syntax, it also provided an architecture and concepts that provide an outline for designing an access control system. This architecture aims to achieve several objectives:

- Ensure effective protection of resources in terms of access control.
- Allow to design an independent system of the used platform.
- Allow the integration of access control system in existing applications.

The overall architecture consists of two main components, each is composed of several components working together. In fact, PPL is a framework that allows the specification, the verification and the implementation of security policies in a modular way. This framework is based on a policy generator and a domain manager. Later in this paper, we detail the design and implementation of these components and their integration into a tool for policies deployment.

Policy generator. PPL defines a policy generator as an interface with the system to secure, it allows to specify, compile and analyse new security policies, and also store and distribute the compiled policies.

Domain manager. Maintains a distributed hierarchy of entities in the domain. Each domain holds references to its entities and policies applying to it. Conceptually, the domain manager works like LDAP (the famous directory service) but with much more elaborate and extended query language.

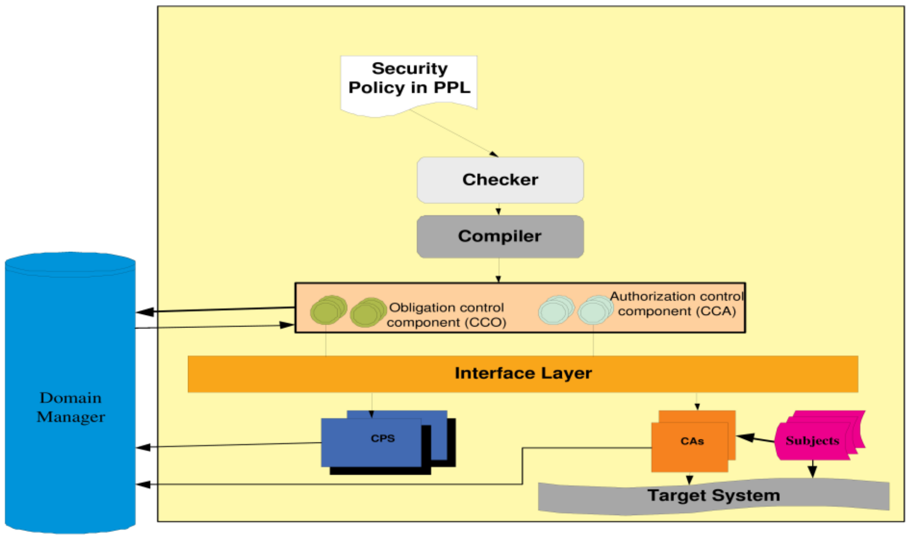

Figure 2: Architecture

4.3. PPL deployment model

The user interferes with the domain manager and with policy generator to structure domain, to specify, to verify, to compile new policies and to save them in the policy store. The policy distribution is established by the policy generator. This includes assigning policies to the correspondent application components and managing dynamic domain membership changes to the entities to which policies are applied.

Since PPL security policy rules explicitly define their objects and subjects, it makes it easier to automatically distribute policies. The obligation policies (positive and negative) are distributed between their subjects and positive and negative authorization policies between their objects. Entities that implement the policies are called application components and can be distributed. The components implementing policies implement an interface for implementation of policies that allows the loading, unloading, activation and deactivation of low-level policies.

The application of an authorization policy is delegated to one or more application components that intercept actions on target objects and check if there is access authorization.

In our model, application components for authorization policies are called access controllers (CAs) and generally interface with lower-level access control mechanisms that actually perform access control, such as firewall.
Subjects of obligation policy are instances of application called component of security policy (CPSs) in which their impacts are defined by the policies applied to them. Thus, the components of security policies directly apply an obligation policy for a subject. CPSs architecture may be generic, but they may be intended for specific applications.

4.4. Policy Generator

4.4.1. Policy compiler

It is a dedicated compiler which can translate specifications written in PPL in the various representations required to implement policies. Figure 3 shows our policies compiler modules. The principal phase of the compilation process is the generation of the intermediate code that feeds the compiler code generators with generated intermediate code. Generated code for a given policy is stored by the code assembler module under the appropriate domain entry in the domain manager.. The default representation matches the filters of fire-wall.

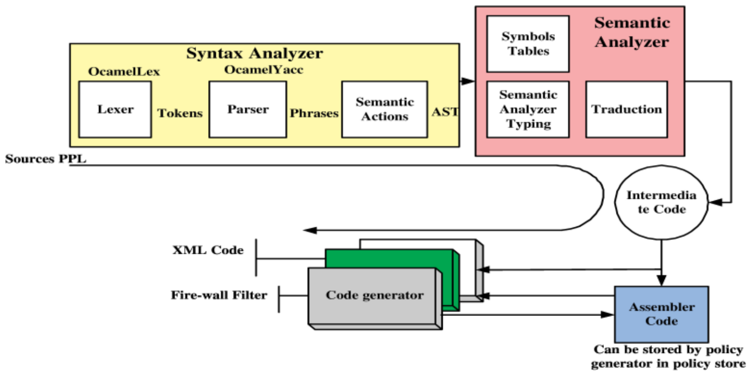

Figure 3: PPL Compiler Architecture

The LALR parser (1) on which our compiler is based, is generated with the ocamlyacc analyzer generator written in ocaml and which is very similar to yacc. The compiler modules are illustrated in Figure 3. The syntax parser, generated from the grammar of the language, analyzes a policy specification in an abstract syntax tree that is then passed to the semantic analyzer.

Semantic analysis begins with the definition of names (scope analysis) and checks that the basic strategies contain all the required elements (completeness checks). Subsequently, the scope analysis is completed, the type analysis is performed and the intermediate code (IC) is generated. The intermediate code is first a clear interface between PPL and the machine. It increases the flexibility of the compiler. It is expressive enough to express all the constructs of language, but also sufficiently close to the actual machines to produce efficient code. An interesting effect of the intermediate code is the unifying impact of its concision.
Various constructions of PPL are expressed by the same construction of the intermediate code, while a construction of the same intermediate code may merge two neighbouring assembler operations. The intermediate code makes it possible to navigate in the specification and to access the elements of the policy, it is a tree of functions corresponding to the specifications of the security policies. The assembly code generator takes the intermediate code as input, and coordinates the code generation phase. The compiler maintains a list of code generators that implement a standard interface and can be dynamically added to the compiler in a plug-and-play way without recompiling the system. This procedure allows the addition of a code generator, requiring minimal intervention from the user and thus minimizing errors in handling and setting. The Code

Assembler module passes the IC to all code generators which are user-activated to generate code. The principal module of the compiler has two interfaces: a standard interface to define the compilation parameters and start the analysis or compilation of a given security policy specification, and an interface to add code generators..

4.4.2. Policy Checker

Policy Checker is the PPL tool to manage and analyze the security policies security in a distributed environment. This tool includes a analyser to perform modality conflict detection, a tool to detect semantic conflicts and validate the policy. Section presents an algorithmic solutions to policy analysis.

4.4.3. The interface layer

The interface layer is customized for specific systems. Its task is to be a bridge between the activities and state of a system and the policy generator. It interprets the natural form of the system and provides the policy generator the system description view. The form of the interface layer depends much on the corresponding system, and there may be different approaches. the interface layer may run over log files and pass activity reports to the generator. In some manner appropriate to the system being protected, the interface layer also needs to determine the proper set of policies to be enforced, and let the policy generator know this.

4.4.4. The components of security policies (CPS)

The components of security policy (CPS) apply all activated obligation policies for a given subject. The components of security policy can serve as a proxy for the user to ensure access to the resources that access is authorized by authorization policies, or act as an automated agent to interpret the obligation policies.
After being created, the Security Policy Components (CPS) are stored in domains under the directory used to specify the subjects of the obligation policies. The figure below provides an overview of how a CPS system works:

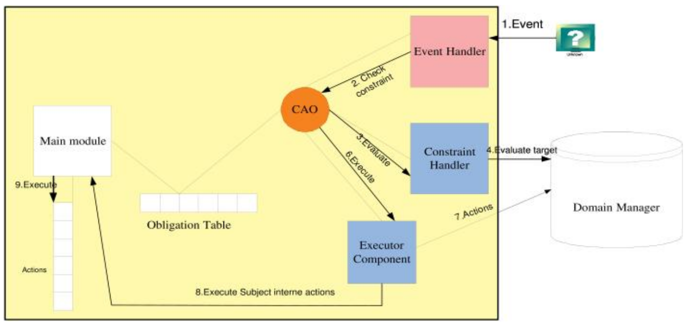

Figure 4: The components of security policy (CPS)

The obligation policy implementations are loaded from corresponding obligation control component (CCO) by an interface provided by the CPS. The appropriate application components encapsulate security policies and store them locally. A bond application component (CAO) encapsulates the bond policies and stores them in the bond policy table. Application component objects are removed from the tables when the corresponding policy is deleted. The operations that make up the policy lifecycle are run on the CPS server using the Security Policy Management Interface. These operations are then passed to the appropriate application components. Once an obligation policy is activated, the corresponding CAO records the specification of the obligation event. CAO is composed of an event manager, a constraint manager and an executor component. The figure above illustrates the steps involved in triggering obligations.

(1) At the reception of an event,
(2) The event manager starts the execution of the obligation policy by launching a request to evaluate the constraint specification.
(3) The constraint manager evaluates the constraint.
(4) If the constraint is based on a target, an evaluation of the constraint must be performed for each of the target objects within the scope of the policy, as this may not be true for all targets. In this case, the constraint manager evaluates the set of target objects of the target domain.
(5) Then move to the constraint evaluation for each target and return all targets for which the policy is valid.
(6) CAO through its executor component, initiates the execution of actions.
(7) Target actions are executed on their associated target sets by executor component,
(8) and the main module of the CPS is called with the name and parameters of the action to execute the subject actions.
(9) Action objects are dynamically loaded into the CPS by the interface that it provides to extend its functionality and to store these objects locally in a table.

4.4.5 The access controllers (CAs)

Authorization policies for one or more target entities are enforced by access controllers. To do this, close interaction with the underlying access control mechanism is required, for example with the host operating system or a firewall, and so on. Access Controllers provide methods for loading, enabling, disabling, and unloading authorization policies. Authorization policies may require different representations depending on the underlying security platform on which the target objects are running. However, the distribution of authorization policies is identical to that described for obligation policies. The Policy generator creates an Authorization Control Component (CCA) corresponding to the Distribution Authorization.
The authorization control component (CCA) evaluates the set of target objects on which the policy is to be distributed, using the authorization policy. The access control to each target object is ensured by the access controller associated with this same object. The entry of the target object in the domain manager references the associated access controller and therefore ensures its access.
The authorization control component (CCA) accesses the access controller reference for each target in the target set of the policy and distributes the policy to all identified access controllers.
In PPL, the implementation of access control relies on the use of enabled authorization policies for configuring the security mechanisms of the underlying platform. Depending on which security platform acts as the access controller, the CCA could additionally request the representation of the policy, stored with the policy entry in the domain manager, to transmit it to the access controllers.
The authorization policy specification is mapped into an appropriate format, which can then be used by the access controllers to configure the underlying security platform. This mapping is done by specialized code generators that can be implemented for this purpose.

4.5. Domain Manager

The domain manager consists of three components, a component for each type of entity that can exist. These are:

- A manager of the components of security policies that manage the changes in the CPS.
- A manager of a target entity that manages the changes in any type of entity except CPS.
- A monitor domain that organizes the entities of the domain by adding or removing the son or a parent entity.

Each of these components involved in implementing a given policy type. Indeed, positive or negative obligation policies are generally concerned by changes in the domains of their

subjects. Therefore, only the manager of CPS and the domain monitor sends notifications of changes to its policies. However, authorization policies depend on changes in the domain of target entities, and therefore only the manager of the target entities and domain monitor are involved.
To complete framework and to meet the remaining implementation issues, we present in the next section the PPL language that allows the specification of a security policy through high-level abstractions.

5. A domain specific language for policy programming

We have designed PPL-Policy Programming Language, a domain-specific language to specify security policies. We began designing our language with a thorough study of the field of computer security in general, and in particular security-based policies. Our domain analysis consisted of the examination of various types of tools, starting with the most well-known specifications of security policies, then the specifications of typical security policies (for example, the IPSec strategy [3]), more specific tools (for example, the Web Services Security Policy [14], the Web Semantic Security Policy [15] and the Security Policy for Clinical Information Systems [5]), we have also reviewed frameworks and tools for the specification of security policy (eg, Ponder2 [6]), as well as various documents and articles.
This domain analysis allowed us to identify the key requirements for a language dedicated to the security domain. The language should have the necessary constructs to describe the appropriate operations performed to ensure the security of a given system.
That's why we built our language around five basic blocks: entity, scope, rule, action and policy. The language must of course include statements specific to each block to allow for specific verifications and analyzes; it must be able to decompose a security policy specification into manageable components, so it must be modular. In addition, policies can also be composed into more complex policies until they form a single comprehensive policy. it should include an interface language to allow disciplined reuse of existing security action libraries.

5.1. Basic concepts of the PPL Language

A PPL program is essentially composed of a list of blocks. Block declarations describe which topics (such as user or process) can access which objects (such as files or devices) and under what condition. A block can be a policy, rule, action, entity or scope. Scope is a representation of the list of entities that may be involved in the policy. The policies correspond to a set of rules for determining specific configuration parameters for some protection of the system; they can be simple or compound. A simple policy is defined by a list of protection actions implemented in another programming language. This feature promotes the reuse of existing action libraries. A composite politics refers to a composition of simple poles. The rule represents a set of constraints on a set of actions; they can be either a single trigger in which a single action is triggered for a given object, or a multiple trigger in which several different actions can be triggered for the same object. The action can be atomic or composed. In the following we detail each of the basic PPL blocks and show how they are used to specify PPL security policies.

PPLSpec	*::=*	*blocks*
blocks	*::=*	*block*
	|	*block*| *blocks*
Block	*::=*	*rule* | *policy*| *action*| *scope*| *entity*

Figure 5: Syntax of a PPL specification

5.1.1. Entity

A PPL entity is a typed object that its properties can be queried by explicit interface. Entity can be an object or a subject or a collection of them. Entity has identification and can be a destination and source of rules.

5.1.2. Scope

Scope is a collection of entities, it is essential in any policy considering that it provides the necessary abstraction to achieve scalability, compactness and generalization. Scopes avoid that each rule to be repeated for each entity to which it applies. Scopes have a name and simplify the management of a large number of entities in a rule. PPL defines two types of scopes: classes and domains. Classes are sets of entities classified according to their properties, e.g. all TCP packets and domains are defined by explicit insertion and deletion of their elements.

policy	::=	*type-pol* policy *ident* ((*params*))? { *policy-def* }
type-pol	::=	pauto | nauto
policy-def	::=	*scope-def body-def* (*constraint-def*)?
scope-def	::=	scope:{ *scope*}
body-def	::=	body:*rules*
rules	::=	*rule*;
	|	*rule*;*rules*
constraint-def	::=	while: *expr*

Figure 6: PPL Syntax of authorisation policy

policy	::=	*type-pol* policy *ident* ((*params*))? { *policy-deleg-def*}
type-pol	::=	pdeleg | ndeleg
policy-deleg-def	::=	*scope-deleg-def body-deleg-def (constraint-def)?*
scope-deleg-def	::=	(subject:{*scope*})?(object:{*scope*})?recipient: {*scope*}
body-deleg-def	::=	body:(*associated-authorisation*)? *actions*
associated-authorisation	::=	auto-policy:*indent*
actions	::=	*action*;
	|	*action*;*actions*
constraint-deleg-def	::=	(while: *expr*)?(cnumber:*type-int*)?

Figure 7:PPL Syntax of a delegation policy

policy	::=	*type-pol* policy *ident* ((*params*))? {*policy-oblig-def* }
type-pol	::=	poblig | noblig
policy-oblig-def	::=	*event-def scope-oblig-def body-oblig-def* (*constraint-def*)?
scope-deleg-def	::=	subject:{*scope*}(object:{ *scope*})?
body-deleg-def	::=	body:*actions*
event-def	::=	event:*expr*
actions	::=	*action;*
	|	*action*;*actions*
constraint-deleg-def	::=	while: *expr*

Figure 8: PPL Syntax of obligation policy

5.1.4. Rule

A rule describes a set of actions that subjects can perform on target objects when a set of constraints is satisfied. A rule is said a *single-trigger*, when condition is satisfied, only one action is triggered. In a *multi-trigger* multiple different actions may be triggered for the same object when condition is satisfied. For example, IPSec crypto-access rule is a single-trigger. In fact, once a traffic matches a certain condition, its action is triggered and no further matching is performed. This is in contrast to crypto-map rules where a particular traffic may match multiple constraints causing multiple actions to be triggered.

rule	::=	rule *ident* ((*params*))? { *rule-def* }
rule-def	::=	{*subjects-def subjects-def constraints-def*}
subjects-def	::=	subject:*entities*
objects-def	::=	object:*entities*
constraints-def	::=	*constraint*
	|	*constraint constraints-def*
constraint	::=	if(*expr*)then *actions*
actions	::=	*action* ;
	|	*action*;*actions*

Figure 9: PPL Syntax of a security rule

5.1.5 Action
Action represents the operation triggered when a constraint is satisfied. Action can be either atomic (protect, bypass, discard) for example in IPSec filtering policy, or composite such as a service implementation.

6 Automating the policy Analysis

6.1 Modality conflicts detection

The precondition for a modality conflict occurs is that policies containing rules using the same subjects, similar actions, the same objects, and the same constraints, take effect at the same period. Therefore, it is necessary to know the time on which a policy will be enforced (for checking inter-policy), and so brought their overlap. The intra-verification of policy seems simple enough. Indeed, the analysis of a policy specification, allows to enumerate all tuples (subject, object, action) on which policy rules are applied. If two or more rules that are applied to a single tuple (subject, object, action), then there is a potential conflict and policy must be checked to see if there is a real conflict (e.g., a rule authorization and a prohibition rule applied to the same tuple (subject, object, action)). A modality conflict can be one of the following types:

6.1.1. Authorisation Conflict
A modality conflict of authorisation occurs when a negative authorisation rule and a positive authorisation rule are defined for the same objects, subjects. The following algorithm is used to detect authorisation conflicts between two rules.

Algorithm 1 tow rules Conflict

```
1: authCon f lictR(R1,R2) : Boolean
2: s1 := GetSubject(R1),s1 := GetSubject (R2)
3: o1 := GetObject(R1),o1 := GetObject (R2)
4: a1 := GetActiont(R1), a1 := GetAction (R2)
5: if ((s1 = s2) and (o1=o2) and (a1 = Deny) and (a2 = permit)) then
6:     TRUE
7: else
8:     false
9: end if
```

The procedure for detecting conflicts between rules is used in a generic procedure which determines all the rules in conflicts in the specification of a policy. It returns an array containing a structure of rules in conflict. Below we present this procedure.

Algorithm 2 Conflict in set of rules

```
1:
2: RS: structure
3: Debut
4: $R1$ : $rule$
5: $R2$ : $rule$
6: Fin
7: Tab_RS: array of structure RS
8: Tab_R: array of rules
9: Tab_P: array of policies
10: authConflict_P (P): Tab_RS
11: $tab1$ : $tabR$
12: $tab2$ : $tabRS$
13: i, j, k, l: integer
14: K =1;
15: $tab1$   get_R (P)
16: **for** ($i = 1; i < (tab1.lenght)-1; i{+}{+}$) **do**
17:     **for** ( $j = i+1; j < (tab1.lenght); j{+}{+}$) **do**
18:         **if** authConflict_R ( $tab1[i]$, $tab1[j]$) = $true$) **then**
19:             $tab2[k].R1 = tab1[i]$ ;
20:             $tab2[k].R2 = tab1[j]$ ;
21:             $K{+}{+}$;
22:         **end if**
23:     **end for**
24: **end for**
```

The authorisation conflicts detection process within the same policy, can be generalized to detect conflicts between different authorisation policies. However, the prerequisite for the occurrence of a modality conflict is that the policies involved hold at the same time. Besides, it is essential to take into account the constraints that control the applicability of the policy. This greatly complicates the conflict detection procedure. To overcome this problem, we define the commence(P) function, which returns the time from which the execution of P policy begins, the finsh(P) function, that returns the time of P policy execution ends, and the constraint(P) which returns the P policy constraint. Below we present the tow policies conflicts detection procedure.

Algorithm 3 tow policies conflicts

```
1: authConflict_interP ($P1$,$P2$, $tab3$,$k$)
2: $tab1$, $tab2$ : $tabR$
3: j, i : integer
4: var c: Boolean;
5: K =1;
6: $tab1$   rename(get_R ($P1$));
7: $tab2$   rename (get_R ($P2$));
8: **if** ((commence($P2$) < finish($P1$)) and ( finish($P2$) > commence($P1$)) and
(constrainte($P1$) =constrainte($P2$)))  **then**
9:     **for** ($i = 1; i < (tab1.lenght); i{+}{+}$) **do**
10:        **for** ( $j = i+1; j < (tab1.lenght); j{+}{+}$) **do**
11:            **if** (authConflict_R ( $tab1[i]$, $tab2[j]$) = true) **then**
12:                $tab3[k].R1 = tab1[i]$;
13:                $tab3[k].R2 = tab2[j]$ ;
```

```
14:             K ++;
15:             C= true;
16:      else
17:        C= false;
18:       end if
19:     end for
20:   end for
21: end if
```

The generalization of this procedure can detect conflicts between different authorisation policies.

Algorithm 4 set of policies conflicts

```
1: authConflict_interP(tab: Tab_P)
2: tab1 : tabRS;
3: k: integer;
4: K =1;
5: for (i = 1; i < ((tab.lenght)−1); i++) do
6:      for ( j = i+1; j < (tab.lenght); j++) do
7:              authConflict_interP (tab[i], tab[ j], tab1,k) ;
8:      end for
9: end for
```

Obligation conflicts This type of conflicts occurs if one policy specifies that a subject is obligated to perform an action when another policy requires that the subject refrain from performing that action. This type of conflict is determined by the following procedure:

Algorithm 5 Obligation conflict

```
1: obligConflict(P1,P2): boolean
2: t1  getType_P(P1);
3: t2  getType_P (P2);
4: S1  getSubjet_P (P1);
5: S2  getSubjet_P (P2);
6: O1   getObjet_P (P1);
7: O2   getObjet_P (P2);
8: A1   getAction_P (P1);
9: A2   getAction_P (P2);
10: if ((commence(P2) < finish(P1)) and (finish(P2)>commence(P1)) and (constraint(P1)
=constraint(P2)) and (t1 = POBLIG) and (t2 = NOBLIG) and (S1 =S2) and (O1 = O2) and
(A1 = A2)) then
11:      true
12: else
13:      false
14: end if
```

Unauthorised Obligation Conflicts This type of conflict occurs if a subject is obliged to perform an operation but there is another policy that prohibits the subject from performing the operation.

Algorithm 6 Unauthorised Obligation Conflicts Detection

```
1: unauthObligConflict(P1, P2): Boolean
2: tab: tab_R;
3: i: integer;
4: t1  getType_P(P1);
5: t2  getType_P (P2);
6: S1  getSubjet_P (P1);
7: O1   getObjet_P (P1);
8: Tab get_R(P2);
9: if ((commence(P2) < finish(P1)) and (finish(P2) > commence(P1)) and (constraint(P1)
   =constraint(P2)) and (t1 = POBLIG) and (t2 = NAUTO) ) then
10:     for (i = 1; i < (Tab.lenght); i++) do
11:       if (((GetSubject(Tab[i]) = S1), (GetObject(Tab[i]) = O1) and GetActiont(Tab[i])
          = DENY)) then
12:             true
13:       else
14:             false
15:       end if
16:     end for
17: end if
```

7 Related Work

Our current work is directly inspired by earlier research on characterization and specification of security policies. We can classify existing security policy specification tools as a following:

- Ponder2: [6] is a declarative, object-oriented language developed for specifying management and security policies. Ponder permits to express authorizations, obligations, information filtering, refrain policies, and delegation policies. Ponder can describe any rule to constrain the behavior of components, in a simple and declarative way. Ponder is certainly the work most related to PPL. While PPL and Ponder share the same goals, their approaches vary considerably. The most difference between PPL and Ponder relates to the relevance to specify security, is the incapacity of Ponder to express certain aspects related to the communication security such as confidentiality and integrity. These aspects are allowed in PPL by the possibility to express the parameters needed for the communications security, for example the source and the destination of a message. Ponder is conceived to be a policy language, however on the level networks, when interactions with standard models of policy is necessary, a rule-based approach is more interesting, because the translation is much easier. For this reason, we conceived PPL in a way to mixer jointly both approaches, indeed PPL provides a flexibility and abstraction in the specification of policies and at the same time allows to specify the rules.
- *Hierarchical Policy Language for Distributed Systems (HiPoLDS):* the authors [7] assume that HiPoLDS was designed to provide a policy specification tool in a concise, readable and extensible manner. The design of HiPoLDS was intended for decentralized environments under the control of multiple stakeholders. It represents the application of the policy through the use of distributed reference controllers, which control the flow of information between services (ie SOAs) and have the ability to implement directives issued by the decision engines. But, unlike PPL, it gives the main directions of implementation, designed for service-oriented architectures, and no reasoning on the scalability and robustness of the proposed solution.

- *JACPoL: A Simple but Expressive JSON-based Access Control Policy Language* [8] [9] Authors claim that JACPoL is simple access control policy language in JSON designed to provide a flexible and fine-grained ABAC (Attribute-based Access Control), and it can be easily customized to express a wide range of other access control models. They have positioned their approach in comparison with the traditional XACML [10] policy language and have tried to show that JACPoL is much lighter, scalable and flexible [11]. The evaluation JACPoL considered only the time required to reply to policy requests. which presents a less realistic scenario and therefore does not provide a better overview of computational resource consumption.
- Policy Management for Standalone Computing (PMAC): [13] is a framework that simplifies the management and automation of complex products and systems through policy-based management. PMAC policies are defined using an XML schema. The structure includes a general constraint specification language that is also specified using XML. PMAC supports policy analysis, debugging and conflict resolution. However, the scanning process does not take into account the behavior of the managed system and therefore cannot detect inconsistencies. These inconsistencies occur when applying a policy results in a state change that causes conflicts between other policies.
- Rei: [12][13] is a policy framework that integrates support for policy specification and analysis. It allows users to express and represent the concepts of rights, prohibitions, obligations and exemptions. In addition, Rei allows users to specify rules defined as rules associating an entity of a managed domain with its set of rights, prohibitions, obligations and exemptions. Rei provides a policy specification language in OWL-Lite that allows users to develop declarative policies on domain-specific ontologies.

Research in conflict analysis has been actively growing over the years, but most of the work in this area addresses general management policies. The authors in [16] [17] focused on identifying modality conflicts by simple analysis between positive and negative authorization security policies and the specification of policy precedence rules in order to resolve conflicts.
In [18] authors have proposed a propositional framework for composing and analyzing access control. This technique was based on a logic-based specification of security policy with a clear semantic that leads to the analysis. This approach is not suitable for identifying causes of conflicts. Among the many approaches to policy specification and analysis, there are a number of proposals for formal, logic-based notations. In particular, based on solid theoretical foundations, the authors in [19] proposed the use of Event Calculus as a specialized first-order logic for formalizing policy specification.
Event Calculus uses familiar notations to specify the system behavior, which can be automatically translated into the logic program representation. Abductive reasoning proof procedures for Event Calculus can be used to detect the existence of potential conflicts in partial specifications and generate explanations for the conditions under which such conflicts may arise.
Although this work offers a promising method to fight the problem of conflict analysis in a generic way, it is not sufficient to provide a complete solution to the problem without meet the needs of an application-specific domain.

8. Conclusion

This paper proposes a framework for specification, verification and implementation of security policy. This framework allows to write security policies modular and independent of the underlying system, based on a set of abstractions. Thus, security policies can be implemented by a developer who does not need to be an expert in computer security. We began by discussing the elements required for such a framework. The deployment model has a significant advantage that is the automatic distribution of

PPL policies to their application components. PPL security policies define subjects and objects (targets), which enables automatic distribution of policies without the need to manually manage the allocation of the policy's implementation components.
Next step is to extend our framework to deal with policy refinement. This area need further work as policies are considered to exist at many different levels of abstraction and the transformation process from high-level policy to low-level implementable has remained a largely unresolved problem.

References

1. Journal Article

[1] Amani Abu Jabal, Maryam Davari , Elisa Bertino , Christian Makaya , Seraphin Calo , Dinesh Verma , Alessandra Russo , Christopher Williams, Methods and Tools for Policy Analysis, ACM Computing Surveys (CSUR), v.51 n.6, p.1-35, February 2019.

[2] Uzunov, Anton V., Eduardo B. Fernandez, and Katrina Falkner. "Security solution frames and security patterns for authorization in distributed, collaborative systems." Computers & Security 55 (2015): 193-234.

[3] Thiruvasagam, Prabhu, et al. "IPSec: Performance Analysis in IPv4 and IPv6." Journal of ICT Standardization 7.1 (2019): 61-80.

[4] Sloman, Morris, and Emil Lupu. "Engineering policy-based ubiquitous systems." The Computer Journal 53.7 (2009): 1113-1127.

[5] Matteo Dell'Amico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana De Oliveira, and Yves Roudier. 2013. HiPoLDS: A Hierarchical Security Policy Language for Distributed Systems. Inf. Secur. Tech. Rep. 17, 3 (February 2013), 81-92.

[6] Jiang, Hao, and Ahmed Bouabdallah. "A JSON-Based Fast and Expressive Access Control Policy Framework." Emerging Technologies and Applications in Data Processing and Management. IGI Global, 2019. 70-91

[7] Hu, C. T., Ferraiolo, D. F., Kuhn, D. R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2019). Guide to Attribute Based Access Control (ABAC) Definition and Considerations [includes updates as of 02-25-2019] (No. Special Publication (NIST SP)-800-162).

[8] Ganek, Alan G., and Thomas A. Corbi. "The dawning of the autonomic computing era." IBM systems Journal 42.1 (2003): 5-18.

[9] Wijesekera, Duminda, and Sushil Jajodia. "A propositional policy algebra for access control." ACM Transactions on Information and System Security (TISSEC) 6.2 (2003): 286-325.

2. Conference Proceedings

[10] Spirakis, Paul, and Philippas Tsigas, eds. Stabilization, Safety, and Security of Distributed Systems: 19th International Symposium, SSS 2017, Boston, MA, USA, November 5–8, 2017, Proceedings. Vol. 10616. Springer, 2017.

[11] Curtis-Black, Andrew, Andreas Willig, and Matthias Galster. "A taxonomy for network policy description languages." 2016 26th International Telecommunication Networks and Applications Conference (ITNAC). IEEE, 2016

[12] Jiang, Hao, and Ahmed Bouabdallah. "JACPoL: a simple but expressive JSON-based access control policy language." IFIP International Conference on Information Security Theory and Practice. Springer, Cham, 2017.

[13] Ferraiolo, David, et al. "Extensible access control markup language (XACML) and next generation access control (NGAC)." Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control. ACM, 2016.

[14] Kagal, Lalana. "Rei: A policy language for the me-centric project." (2002)

[15] Kagal, Lalana, Tim Finin, and Anupam Joshi. "A policy language for a pervasive computing environment." Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks. IEEE, 2003.
[16] Zhao, Hang, Jorge Lobo, and Steven M. Bellovin. "An algebra for integration and analysis of ponder2 policies." 2008 IEEE Workshop on Policies for Distributed Systems and Networks. IEEE, 2008.
[17] Russello, Giovanni, Changyu Dong, and Naranker Dulay. "Authorisation and conflict resolution for hierarchical domains." Eighth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07). IEEE, 2007.
[18] Zhao, Hang, Jorge Lobo, and Steven M. Bellovin. "An algebra for integration and analysis of ponder2 policies." 2008 IEEE Workshop on Policies for Distributed Systems and Networks. IEEE, 2008.
[19] Bandara, Arosha K., Emil C. Lupu, and Alessandra Russo. "Using event calculus to formalise policy specification and analysis." Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks. IEEE, 2003.
[20] Hamdi, Hedi, Adel Bouhoula, and Mohamed Mosbah. "A software architecture for automatic security policy enforcement in distributed systems." The International Conference on Emerging Security Information, Systems, and Technologies (SECUREWARE 2007). IEEE, 2007.
[21] Hamdi, Hedi, Mohamed Mosbah, and Adel Bouhoula. "A domain specific language for securing distributed systems." 2007 Second International Conference on Systems and Networks Communications (ICSNC 2007). IEEE, 2007.

Authors

Author's Name, Author's profile.

Dr. Hedi HAMDI received his PhD in Computer sciences from The University of Bordeaux France in December 2009. Currently, he is an Assistant Professor at Computer Science Department, College Of Computer and information Science, Jouf university where he is conducting research activities in areas of cloud computing, information security, software defined network, Network Functions Virtualization. Dr. Hedi HAMDI is a member of RIADI GLD Laboratory (ENSI Manouba University).