Formal Characterization and Automatic Detection of Security Policies Conflicts

Hédi Hamdi
Manouba University, ESC
Campus Universitaire de la Manouba
2010 Manouba, Tunisia
Email: hamdi.h@gmail.com

Abstract—**Policies, which are widely deployed in networking services (e.g., management, QoS, mobility, etc.), are being a promising solution for securing wide distributed systems and one of the most actual directions of research in the information security area. However, Policy-based security may involve interactions between independent decision making components which may lead naturally to inconsistencies, a problem that has been recognized and termed as policy conflict. Work on policy analysis has mainly focused on conflicts that can be determined statically at compile time. Using formal methods, with good tool support, to express the policies, can not only support the detection, but also help all the involved actors in understanding and resolving the conflicts. The main focus of this paper is on giving a theory and automated techniques for discovering common types of security policy conflicts.**

Keywords-Policies; Distributed systems; Conflicts; Detection.

I. INTRODUCTION

Policies, which are extensively deployed in networking services (e.g., management, QoS, mobility, etc.), are being praised as promising solution for securing widely distributed systems and could also be considered as one of the most recent directions of research in the information security field. However, several problems remain to be solved in this field. One interesting problem of policy based security is how to detect conflicts in a security policy specified for a network behavior. In fact, deploying a conflicting policy within a network is often the origin of unexpected damage. For this reason and once policies are specified and before they are enforced, it should be possible to determine that there are no conflicts between components of the policy. Previous works on issue of policy conflict detection have mainly focused on conflicts that can be determined statically at compile-time [12]. The detection process involved simple policy analysis. Although we believe that static analysis is very useful for detecting some conflicts before policies are deployed, it cannot detect many conflicts in resource management policies which occur as a result of the current state of the resources. For example, policies which increment or decrement allocation of resources may conflict with policies related to setting upper and lower bounds for the resources. These conflicts result from current state of the resource allocation and bounds so can only be detected at run-time. This paper focuses on the PPL (Policy Programming Language) [5][6] a domain specific policy language with a powerful dynamic semantic and with available Software tools, based on which techniques for aiding policy analysis and refinement can be developed.

The work presented in this paper addresses the shortcomings in previous work in the field. It defines a formal model to deal with a range of conflicts in security policy, and an algorithmic solution to facilitate automation of the analysis process.

This paper begins by stating the problem of conflict detection in Section 2, followed by a presentation of our policy model in Section 3. Section 4 introduces a formal model for conflicts detection in security policy. Section 5 presents an algorithmic solutions to the automation of the analysis process and Section 5 concludes the paper and discusses future Works.

II. PROBLEM STATEMENT

In recent years, the trend in the software industry has been directed towards the development of software that can be customized by the user to meet their individual needs. In this context, policies are a very useful way in which the customization can be delivered. Policies also separate the behavioral aspect of software and its main functions. This allows either the main functionality of the software or custom user's behavior to be changed without affecting the other [12]. In a given system, there may coexist multiple policies, it is important to consider how one policy will affect another. Policies that are triggered at the same time, and they contradict each other, are said in conflict. The process of checking policies to see if they conflict is called policy analysis, and conflicts can be detected at specification time. By Lupu and Sloman [12], two types of conflicts can occur between policies: modality conflicts and semantic conflicts (also called application specific). In their work, they associated policies with a mode. According to their definition, a modality conflict arises when two policies with opposite modality refer to the same subject, actions, and objects. This can happen in three ways:

- The subjects are both obligated-to and obligated-not to perform actions on the objects.
- The subjects are both authorized and forbidden to perform actions on the objects.
- The subjects are obligated but forbidden to perform actions on the objects.

 ISBN: 978-1-61208-231-8

Application specific conflicts occur when two rules contradict each other due to the context of the application. We use for our policy model, the PPL (Policy Programing Language) [5][6], our policy specification language (PSL) that appears to be the most flexible. It offers both positive and negative modifications of authorization and obligation policies. Although we do not focus on policy entry in this paper, we assume that all policies entered into the system are done so in the form of PPL language.

III. OUR POLICY MODEL

A. PPL: The Policy Programming Language

We use a domain-specific, special-purpose language to express security policies. The design of our language has been guided by a thorough study of the domain of computer security in general, and especially security based policies. We examined various kinds of tools mainly including specifications of known security policies, specification for typical security policy (e.g., IPSec policy [13]) as well as more dedicated ones (e.g., web services security Policy [10], Security Policy for web semantic [8] and Security Policy for Clinical Information Systems [1]), frameworks and tools for security policy specification (e.g., Ponder [3], PDL [11]), and various documentations and articles. Based on this domain analysis, we have identified the following key requirements for a language dedicated to this domain. The language should be composed of five basic blocks: entities, scopes, rules, actions and policies to describe the appropriate operation performed for ensuring security of a given system; it should include block-specific declarations to enable dedicated verifications and analysis to be performed; it should be modular to enable a security policy specification to be decomposed into manageable components and also policies can also be composed into more complex policies until it forms a global and single policy; it should include an interface language to enable disciplined re-use of existing security actions libraries.

B. Core Concepts of PPL

A PPL program essentially defines a list of blocks. Blocks declarations describe which subjects (e.g., users or processes) may access which objects (e.g., files or peripheral devices) and under which circumstances. A block can either be a policy, a rule, an action or an entity, a scope. Scope represents a list of entities involved in policy. Policies correspond to a sequence rules to determine specific configuration settings for some protection of system; they can be either simple or compound. A simple policy refers to a list of protection action implemented in some other programming language. This facility enables existing actions libraries to be re-used. A compound policy is defined as a composition of simple policies. Rule consists of a set of constraints on a set of actions; they can be either a single-trigger where only one action is triggered for a given object or a multi-trigger where multiple different actions may be triggered for the same object. Action can be atomic or compound in the following we present in details each of the basic blocks comprising PPL and show how they are used in writing PPL security policies.

```
PPLSpec    ::= blocks
Blocks     ::= block j block blocks
Block      ::= rule | policy| action| scope| entity
```

Fig. 1. Syntax of a PPL specification

1) Entities

PPL entities are typed objects with an explicit interface by which their properties can be queried. Entities can be an object or a subject or a collection of them. They have identification and can be a source and a destination of rules.

2) Scopes

Entities can be collected into Scopes. Scopes are essential in any policy considering that they provide the necessary abstraction to achieve compactness, generalization and scalability. Without Scopes, each rule has to be repeated for each entity to which the rule applies. Scopes have a name and they are used in rules for simplified management of large numbers of entities. PPL offers two types of scopes: classes and domains. Classes are sets of entities that are classified according to their properties e.g., all TCP packets, and domains are sets defined by explicit insertion and removal of their elements.

3) Policies

A PPL policy is a group of rules and scopes that govern particular domain events. These rules are used to define the right behavior of a system. PPL supports an extensible range of policy types:

- Authorization policies are essentially security policies related to access-control and they specify whether a sequence of actions, a subject, is permitted or forbidden to perform to a set of target objects. They are designed to protect target objects, so they are interpreted by access control system.

```
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
```

- Obligation policies specify what a sequence of actions, a subject must perform to a set of target objects, on response to particular events and define the duties of subjects in scope of policy. Obligation policies are normally triggered by events.

 ISBN: 978-1-61208-231-8

```
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
```

- Delegation policies specify which actions and rights subjects can delegate to others. A delegation policy thus specifies an authorization to delegate.

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

- Compound policies are used to group a set of related policy specifications within a syntactic scope with shared declarations in order to simplify the policy specification task for large distributed systems.

4) Rules

A rule consists of a set action that subjects can perform on target objects when a set of constraints are satisfied. In single-trigger rule, only one action is triggered when condition is satisfied. 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 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
```

5) Actions

Actions represent the operations triggered when a constraint match. Actions can be either atomic for example in IPSec filtering policy, actions are protect, bypass, discard, or composite such as a service implementation.

IV. FORMAL MODEL OF CONFLICTS DETECTION

There are two broad categories of policy conflicts namely static and dynamic conflicts. As conflict detection can be computationally intensive, time consuming and therefore costly and would preferably be done statically, at compile time. Identified static conflicts therefore require immediate attention, as it will most certainly result in a conflict at some time. Whereas the transformation of a potential, dynamic conflict in a real conflict is quite unpredictable; that is, the inconsistency may be exposed temporarily, or indeed not all. The main purpose of conflict detection is:

- The identification of actual conflict that has occurred and can be resolved statically, at compile-time.
- The prediction of a conflict, that may, occur in the future (and more specifically, exactly what circumstances will expose that conflict)

It should be noted, however, that all the predicted conflicts require notification or action. In some cases, for example, conflict can be predicted to occur, but be far enough into the future or uncertain enough that an alert has no real importance and action would be inappropriate at present. To be able to detect conflicts statically and at runtime, one must know the temporal characteristics of policies in the specification. To do that, we define $start_{t/e}(P)$ that refers to time/event start attribute of the policy, $finish_{t/e}(P)$ that refers to time/event finish attribute of the policy, $recurt_{t/e}(P)$ that refers to time/event recur attribute of the policy and *constraint(P)* which returns the constraint of the policy.

A. Static conflicts detection

1) Authorization conflicts

a) Conflict between two policy rules

□ $(Subject(R_x) \cap Subject(R_y) \neq \phi) \wedge (Object(R_x) \cap Object(R_y) \neq \phi) \wedge (Action(R_x) = DENY) \wedge (Action(R_y) = PERMIT) \rightarrow authoConflict(R_x;R_y)$

Conflicts in a set of policy rules

Let R a set of security rules, $|R|$ the cardinality of R

□ $0 \leq i \leq |R|;\ \forall 0 \leq j \leq |R|$
$\exists R_i;\ R_j \in R \wedge authoConflict(R_i;R_j) \rightarrow RConflict(R)$

b) Conflicts between two policies

We denote a PPL policy by *P(S; B; C),* where S designates the scope of *P* policy, *B* indicates its Body and *C* the policy constraint. The detection of authorization conflicts process within the same policy, can generalized to detect conflicts in a set of authorization policies. However, it is necessary to

ensure that these policies have equivalent constraints (if it is defined, the specification of the constraint in an authorization policy is optional in PPL), they must run in the same period.
Let $P_1 (S_1, B_1, C_1)$, $P_2(S_2, B_2, C_2)$ two authorization policies. P_1 and P_2 are in conflict:

- $(start_t(P_2) < finish_t(P_1)) \wedge (start_t(P_1) < finish_t(P_2)) \wedge (constraint(P_1) \equiv constraint(P2)) \wedge RConflict(B_1 \cap B_2) \rightarrow PConflict(P_1,P_2)$

2) Obligation conflicts

- $\forall start_t(P_2); finish_t(P_2); recurt(P_1); start_t(P_2) < recur_t(P_1) < finish_t(P_2) \rightarrow PConflict(P_1;P_2)$

- $\forall start_t(P_2); recur_t(P_1); finish_e(P_2) (start_t(P_2) < recur_t(P_1) < finishe(P_2)) \rightarrow PConflict(P_1;P_2)$

B. Dynamic conflicts detection

- $\forall finish_t(P_2); Recur_t(P_1); start_e(P_2) (start_e(P_2) < recur_t(P_1) < finish_t(P_2)) \wedge (trigger(start_e(P_2))) \rightarrow PConflict(P_1;P_2)$

- $\forall recur_t(P_1); start_e(P_2); finish_e(P_2) (start_e(P_2) < recur_t(P_1) < finish_e(P_2)) \wedge (trigger(start_e(P_2))) \rightarrow PConflict(P_1;P_2)$

- $\forall start_t(P_2); finish_t(P_2); recur_e(P_1) (start_t(P_2) < Recur_e(P_1) < finish_t(P_2)) \wedge (trigger(start_t(P_2))) \rightarrow PConflict(P_1;P_2)$

- $\forall start_t(P_2); finish_e(P_2); Recur_e(P_1) (start_t(P_2) < recur_e(P_1) < finish_e(P_2)) \wedge (trigger(start_t(P_2))) \rightarrow PConflict(P_1;P_2)$

- $\forall finish_t(P_2); start_e(P_2); Recur_e(P_1) (start_e(P_2) < recur_e(P_1) < finish_t(P_2)) \wedge (trigger(start_e(P_2))) \rightarrow PConflict(P_1;P_2)$

- $\forall start_e(P_2); finish_e(P_2); recur_e(P_1) : event_j start_e(P_2) < Recur_e(P_1) < finish_e(P_2) \wedge (trigger(start_e(P_2))) \rightarrow PConflict(P_1;P_2)$

V. AUTOMATING THE POLICY ANALYSIS

A. Modality conflicts detection

The precondition for a modality conflict occurs is that policy 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 enumerating 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:

1) Authorization Conflict

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

Algorithm 1: two rules Conflict

```
authConflict_R(R1;R2) : Boolean
begin
        s1 := GetSubject(R1),s1 := GetSubject (R2)
        o1 := GetObject(R1),o1 := GetObject (R2)
        a1 := GetActiont(R1), a1 := GetAction (R2)
        if ((s1 = s2) and (o1=o2) and (a1 = DENY) and
        (a2 = PERMIT)) then
                TRUE
                else
                FALSE
end
```

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

```
begin
        RS:structure
        begin
                R1 : rule
                R2 : rule
        end
        Tab_RS: array of structure RS
        Tab_R: array of rules
        Tab_P: array of policies
        authConflict_P (P): Tab_RS
        tab1 : tabR
        tab2 : tabRS
        i, j, k, l: integer
        K =1
        tab1 ← get_R (P)
```

```
        for (i = 1; i < (tab1.lenght) - 1; i + +) do
        for (j = i + 1; j < (tab1.lenght); j + +) do
                if(authConflict_R(tab1[i];tab1[j])=
                TRUE) then
                tab2[k]:R1 = tab1[i]
                tab2[k]:R2 = tab1[j]
                K + +;
end
```

The authorization conflicts detection process within the same policy can be generalized to detect conflicts between different authorization 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 *finish(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 two policies conflicts detection procedure.

Algorithm 3: two policies conflicts

```
authConflict_interP (P1; P2; tab3; k)
begin
        tab1; tab2 : tabR;
        j, i : integer;
        var c: Boolean
        K =1;
        tab1 ← rename(get_R (P1));
        tab2 ← rename (get_R (P2));
        if ((commence(P2) < finish(P1)) and ( finish(P2)
        > commence(P1)) and (constraint(P1)
        =constraint(P2))) then
                for (i = 1; i < (tab1.lenght); i + +) do
                        for (j = i+1; j < (tab1.lenght); j
                        ++) do
                        if (authConflict_R ( tab1[i];
                        tab2[j]) = true) then
                        tab3[k]:R1 = tab1[i];
                        tab3[k]:R2 = tab2[j] ;
                        K + +;
                        C= TRUE;
                        else
                        C= FALSE;
end
```

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

Algorithm 4: set of policies conflicts

```
authConflict_interP(tab: Tab_P);
begin
        tab1 : tabRS;
        k: integer;
        K =1;
        for (i = 1; i < ((tab.lenght) - 1); i + +) do
                for (j = i + 1; j < (tab.lenght); j + +) do
                authConflict_interP (tab[i]; tab[j]; tab1; k);
end
```

2) Obligation conflicts

This type of conflicts occurs if one policy specifies that a subject is obliged 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

```
obligConflict(P1; P2): boolean
begin
        t1 ← getType_P(P1);
        t2 ← getType_P (P2);
        S1 ← getSubjet_P (P1);
        S2 ← getSubjet_P (P2);
        O1 ← getObjet_P (P1);
        O2 ← getObjet_P (P2);
        A1 ← getAction_P (P1);
        A2 ← getAction_P (P2);
        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
                TRUE
                else
                FALSE
end
```

a) Unauthorized 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: Unauthorized Obligation Conflicts Detection

```
unauthObligConflict(P1, P2): Boolean
begin
        tab: tab_R;
        i: integer;
        t1 ← getType_P(P1);
        t2 ← getType_P (P2);
        S1 ← getSubjet_P (P1);
        O1 ← getObjet_P (P1);
        Tab ← get_R(P2);
```

if ((commence(P_2) < finish(P_1)) and (finish(P_2) > commence(P_1)) and (constraint(P_1) =constraint(P_2)) and (t_1 = POBLIG) and (t_2 = NAUTO)) **then**
for (i = 1; i < (Tab:lenght); i + +) **do**
if (((GetSubject(Tab[i]) = S_1), (GetObject(Tab[i]) = O_1) and GetActiont(Tab[i]) = DENY)) **then**
TRUE
else
FALSE
end

VI. Related Work

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 [12] 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. Jajodia [7] has proposed a technique based on deductive reasoning to policy analysis. This technique used 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 [9], the authors in [2] proposed the use of Event Calculus as 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. Adductive reasoning proof procedures for Event Calculus [4] 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 solve 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.

VII. Conclusion

In this paper we have presented a formal characterization of security policy analysis, together with algorithmic solutions to policy analysis. To support automation of conflict detection for security policy, we first defined security conflicts in a formal way. Then, we developed mechanisms to systematically detect conflicts. Our work for policy analysis has been tested through the development of a prototype implementation. Next step is to extend our formalism 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] R. J. Anderson. A security policy model for clinical information systems. In 1996 IEEE Symposium on Security and Privacy, pages 30–42. IEEE Computer Society Press,

[2] Bandara, E. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2003), pages 26, 2003.

[3] N. C. Damianou. Policy Framework for the Management of Distributed Systems. PhD thesis, Imperial College. London, U. K., February 2002.

[4] M. Denecker and A. Kakas. Abduction in logic programming. Handbook of Logic in Artificial Intelligence and Logic Programming, 5:235–324, 1998.

[5] H. Hamdi, M. Mosbah, and A. Bouhoula. A domain specific language for securing distributed systems. In. ICSNC '07 Proceedings of the Second International Conference on Systems and Networks Communications Page 76, 2007.

[6] H. Hamdi, M. Mosbah, and A. Bouhoula. A declarative approach for easy specification and automated enforcement of security policy. International Journal os Computer Science and Networks, 8(2):60–71, Feb 2008.

[7] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In 18th IEEE Computer Society Symposium on Research in Security and Privacy, pages 31–42, 1997.

[8] L. Kagal, T. W. Finin, and A. Joshi. A policy based approach to security for the semantic web. In International Semantic Web Conference, pages 402–418, 2003.

[9] R. Kowalski and M. Sergot. logic-based calculus of events. New Generation Computing, 4:67–95, 1986.

[10] Lalana Kagal, Massimo Paolucci, Naveen Srinivasan, Grit Denker, Timothy W. Finin and Katia P. Sycara: Authorization and Privacy for Semantic Web Services. IEEE Intelligent Systems, pages 50-56, 2004.

[11] J. Lobo, R. Bhatia and S. Naqvi. A policy description language. In Proc. AAAI '99/IAAI '99, Pages 291-298 July, 1999.

[12] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. IEEE Trans. Softw. Eng., 25(6):852–869, 1999.

[13] G. Wolfien. Network ipsec specification. In Pan-European Harmonisation of Vehicle Emergency call Service Chain.Information Society Technologies (IST), 25. October 2002.

 ISBN: 978-1-61208-231-8