Policy Languages Require the Same Composition Mechanisms as Programming Languages

Tom Dinkelaker
Technische Universität Darmstadt
dinkelaker@cs.tu-darmstadt.de

Sascha Hauke
Technische Universität Darmstadt
sascha.hauke@cased.de

ABSTRACT
Current policy languages come with a monolithic syntax and support only a limited set of security formalisms. Thus, contemporary policies can only inadequately prescribe the correct behavior of a distributed business application w.r.t. different views, such as usage control, safety properties, or governance. To support composing policies that involve multiple views, we propose to include well-established composable mechanisms into policy languages. In this paper, we propose an extensible security DSL that composes multiple mechanisms—namely inheritance, scoping, aspects, and different paradigms—into one composite policy language.

Categories and Subject Descriptors
D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications—Extensible languages, Multiparadigm languages

General Terms
Languages, Policy Languages, Inheritance, Scoping, Aspects

1. INTRODUCTION
Today, distributed systems assume a homogeneous security infrastructure that uses a monolithic policy language that typically is a non-Turing complete and domain-specific language (DSL). Often a policy language uses a concrete DSL syntax, it comes with a fixed set of security primitives that are part of a specific security formalism (e.g. RBAC or security automata), and that follow a certain paradigm (e.g. rule-based or state machines). In the policy language, developers specify policies that define part of the correct expected behavioral of the system. It supports only one enforcement mechanism specialized for only one particular purpose, but it is isolated from other policy languages and their enforcement mechanisms. Nonetheless in a globalized economy, distributed systems begin to span multiple applications, stacks, companies, and markets, in which different policy languages and enforcement mechanisms are used. To freely compose policies over multiple views of the correct behavior, such usage control, safety properties, or governance, in this paper, we propose an extensible policy language that leverages well-established composition mechanisms from existing general-purpose programming languages, namely inheritance, scoping, aspects, and paradigms.

The contribution of the paper can be seen in two ways. On the one hand, it provides a requirement analysis for a policy language for distributed business applications. On the other hand, it proposes a preliminary language design that integrates well-established composition mechanisms into a policy language. Although there is yet no concrete implementation, the paper sketches what techniques from programming languages we will use to implement these mechanisms.

In the remainder of the paper, Section 2 defines the special requirements for policy languages in distributed business applications and discusses problems of current policy languages. Section 3 proposes a set of composition mechanisms that we think are essential for enabling freely composable security policy languages. Section 4 concludes the paper.

2. INSUFFICIENCIES IN POLICIES FOR DISTRIBUTED BUSINESS APPLICATIONS
Distributed business applications comprise various loosely-coupled software components in a SOA that run in different systems, stacks, companies, and markets. In the following, we discuss a set of example policies for such applications which require special mechanisms that are currently not completely supported in contemporary policy languages.

2.1 Static Policies Lack Semantic Flexibility
In a globalized economy, processes involve international business partners. Corresponding applications combine services and providers distributed over different countries. For the execution of such services, the providers must take into account differing national legislation which individual services and combined processes are obliged to adhere to. Consequently, a policy defined for an international business application needs to be adapted to its execution context.

Consider the WSPL policy in Listing 1 that enumerates a set of payment options available at the location a service request originates from. Say if a request originates from a foreign country, the policy accepts only <pre-paid> orders. In contrast for domestic requests, all options are available, such as <credit card payment, bank invoice> as well as <pre-paid>.

With current policy languages, such as WSPL, it is possible to specify such a policy but only in a hard-wired way. For example, such a policy could use a conditional state-
2.2 Nested Policies Lack a Precise Scoping

In WS-Policy, a policy expression that is an element of another policy expression is called a nested policy. The nested policy enriches the enclosing policy by defining further details. For example, in Listing 2, the nested policy (lines 5–11) defines that the enclosing policy (lines 1–11) for the symmetric transport binding (line 2) must use a Kerberos token.

WS-Policy enables defining security capabilities and requirements for single Web service. Now consider a service composition with a global policy affecting multiple system layers or levels in an enterprise environment. In this context, policies need to provide flexible means to define which parts of the policy affect what service components or enterprise levels. However, policy languages that only support static policies for single end points, such as WS-Policy, do not support precisely scoping their effect to parts of the system.

2.3 Policies Suffer from Tangling

Just as in programming languages, crosscutting concerns exist in policy languages. Such concerns may comprise, for instance, demands on auditing and logging of policy enforcement and monitoring, transport layer or storage encryption, data and information flow control, which are potentially shared among differing policy rules governing service execution or data storage. Spreading out requirements affecting various rules in a policy violates the principle of the separation of concerns – similar to violating encapsulated code in OO programming languages. This leads to tangled and also scattered policy specifications, which makes maintenance of policies complicated, and hinders developers and users to quickly comprehend a policy’s content and intention.

Listing 2: A WS-Policy policy with a nested policy

```
Policy (Id = "Service Levels") {
  Rule {
    Location = "Germany", Fee = 5, Currency = "EUR",
    Options = { "Pre-Paid", "Credit-Card", "Invoice" }
  }
  ...
  Rule {
    Location = "USA", Fee = 7, Currency = "USD",
    Options = { "Pre-Paid" }
  }
}
```

Listing 3: A Ponder policy defining a policy group

```
Listing 4: A tangled and scattered policy

```

```
```

```
```
the corresponding rules. Policy assertions relating to different security concerns are tangled in one rule, such as the severity level for privacy (line 6) and the encryption algorithm for confidentiality (line 7). Policy assertions relating to one security concern are scattered over multiple rules, such as the encryption algorithm BlowFish (lines 7 and 16). Tangling and scattering leads to poor maintainability of the policy specification due to crosscutting concerns.

What is needed for modularizing tangled policies are language mechanisms that help to encapsulate and disentangle non-functional concerns in a clear and concise way.

2.4 Policies Lack Integrating Formalisms

Policies express different goals. Like these goals vary, so do their representations in policy languages that specify them. Generally, different policy languages do not share a common syntax, even if they are based on the same fundamental representation technology, such as XML (e.g. WS-Policy) or EBNF (e.g. Ponder 2). Furthermore, policy statements are adapted to the concepts underlying the specification paradigms, with each policy supporting a single paradigm, and thus formalism. This limited view is insufficient when dealing with distributed business applications. A holistic specification of the system’s behavior requires a combination of views of the complete system. Because different representations and formalisms have various degrees of power regarding the description of these views, it is difficult to determine a single, ideal language suited to their expression.

What is needed for policies are methods for integrating different paradigms addressing different views of the system.

3. OVERVIEW OF RELATED WORK

Current policy languages only support one concrete syntax that supports only a limited set of security formalisms and paradigms. Existing languages are not open for new syntax and semantics, which is what new formalisms and paradigms would require. They do not support policies like those discussed in Section 2.4 because they all share the insufficiencies that policies are not semantically flexible, not precisely scoped, contain scattered and tangled code fragments, and adhere to one paradigm only.

Ponder 2 is an object-oriented policy language with a very basic support for controlling the policies in policy groups, but particularly it neither supports full-fledged lexical nor dynamic scoping.

WS-Policy 1 is a XML-based policy language framework that allows integrating new domain-specific policy languages with an XML-based syntax, but in particular concrete syntax is not supported.

XACML 2 is an XML-based and rule-based policy language for defining attributes the specification of authorization policies and obligations. Specifically, but policies for a behavioral specification of a component (e.g. security automata) are out of scope.

What is needed is an extensible policy language that users can tailor for the specific requirements of their business application. So that, for such a policy language, they can select the right concrete DSL syntax, composition mechanisms, security formalisms, or paradigms that they want to use, to be included into their policy language.

4. INTRODUCING COMPOSITION MECHANISMS INTO POLICY LANGUAGES

In order to overcome the limitation of existing policy languages discussed in Section 2, we propose to make composition mechanisms from programming languages available in policy languages. Specifically, we consider adapting polymorphism and scoping for meeting the need of policy language for semantic flexibility. Furthermore, we propose the use of aspects for addressing scattered and tangled policy definitions, and an open set of paradigms to define policies with the right syntax and semantics. In the following, we present example solutions in WSPL, however WS-Policy or another policy language could be extended with the same mechanisms in a similar manner.

4.1 Polymorphy

To enable flexible policies, we propose to extend the policy language with an inheritance mechanism that is similar to polymorphic programming languages. The inheritance mechanism enables the end users to refine the rules of a base policy in an extended policy.

```
1 Policy (Id = "Service−Levels") { Rule { Trust = "High", Options = { "Pre−Paid", "Credit−Card", "Invoice" } } Rule { Trust = "Low", Options = { "Pre−Paid" } } }
```

Listing 5: Base of a polymorphic policy

For example, Listing 5 and Listing 6 show two modular policies, where the latter policy extends the former – like a subclass extends its super class. Listing 5 shows the base policy Service−Levels that defines the payment options for different trust levels. Listing 6 shows the policy extension Specific−Service−Levels that defines what Low trust or respectively High trust means. Note that, even when different stakeholders define those policies at different times or in different sub-systems, the policy extension Specific−Service−Levels can refine what Low or High trust means for the Service−Levels base policy.

```
1 Policy (Id = "Specific−Service−Levels") { 2 Extends (Super = "Service−Levels") { Rule { Location = "Germany", Experience−Level = "Good", Experience−Length = "Long", Trust = "High" } Rule { Location = "USA", Platform−Monitor = "Deployed", Trust = "High" } 4 Rule { Trust = "Low" } 6 }
```

Listing 6: Extension of a polymorphic policy

We expect that having an inheritance mechanism available in policy languages, we can provide policy developers with similar advantages as having OO inheritance w.r.t. extensibility, reusability, and modular reasoning. However, it is a challenge to provide such an inheritance mechanism for an open set of policy dialects. On the one hand, the inheritance mechanism need to define a default polymorphic semantics that allows to refine policies by overriding parts of them at
the level of assertions. On the other hand, the inheritance mechanism needs to be extensible for special cases in which it must take into account the specific semantics of a policy dialect. While we expect that end users can use the default polymorphic semantics for most cases, only an extensible inheritance mechanism enables domain-specific composition semantics for composition.

4.2 Scoping

To precisely scope policies, we propose to support different scoping schemes in the policy language. Every scope describes a partial view of the system that is structured in different topologies, such as an organizational or a technical topology, of which several topological views can overlap. To define a new scope, there is a special operator Scope with: (1) an Id that defines a unique identifier for the scope within a topology, (2) a scoping Strategy that defines how the defined elements in its body propagate, and (3) a Priority for resolving conflicts between overlapping scopes. Inside the scope operator, a nested policy defines a binding for that policy in this scope.

Depending on the topological view and the scoping strategy, the contained bindings can propagate to other scopes or parts of the system. With lexical scoping, elements of an enclosing scope propagate to its nested scopes. With dynamic scoping, the definition of an element always establishes a new binding that propagates globally through all scopes.

For example, Listing 7 shows several policies that are nested inside different scopes. The corresponding Scope operators select the right scope and strategy for the nested policies that all have the same Id. Since there are different scopes defined for the policies, there is no name clash between them. The first two scopes allow defining the Key-Length policy differently for the two companies Organizational.MarketX.Company1 with 1024 bits (lines 3–9) and Organizational.MarketX.Company2 with 512 bits (lines 10–14). Because Company1 uses a lexical scoping strategy, it is possible to redefine the policy within a certain department (e.g., Dep1 with 2048 bits). There is another scope that defines a special Key-Length policy for devices with limited resources. There are two dynamic policies that impose restrictions on the maximum key length, namely 256 bits when using the algorithms to communicate with sensor nodes, and 128 bits when the systems detects that the battery of a mobile sensor device is low. Each scope implicitly binds the policies that are nested into it body.

Alternative, one can explicitly define the scope of a policy by using the policy operator’s optional Scope attribute. For example, the policy in line 7 redefines the Key-Length policy. Because the policy is explicitly scoped to Organizational.MarketX.Company1.Dep1 through the Scope attribute of the policy, however this only overrides the binding of this policy within the department Dep1 of Company1.

When scopes of different topologies overlap, there can be multiple bindings for one policy Id. Consider a sub-component that is part an organizational topology element Company1 and part of a technical topology Technical.NetworkA.SensorNodes. In Listing 7, there are different policies defined for this sub-component by the two scopes Organizational.MarketX.Company1 and Technical.NetworkA.SensorNodes. Therefore, it is necessary to resolve such conflicting bindings. To resolve such conflicts, we always select the binding from the scope with the higher priority.

### Listing 7: A policy using different scoping schemes

```plaintext
1 Scope (Id = "Organizational.MarketX", Strategy = "Lexical", Priority = "Normal") {  
2 Scope (Id = "Company1", Strategy = "Lexical") {  
3 Policy (Id = "Key-Length") {  
4 Rule (  
5 Key-Length = "1024-bit"
6 ) } }  
7 Policy (Id = "Key-Length", Scope = "Dep1") {  
8 Rule ( Key-Length = "2048-bit" )  
9 }  
10 Scope (Id = "Company2") {  
11 Policy (Id = "Key-Length") {  
12 Rule {  
13 Key-Length = "512-bit"
14 ) } }  
15 Scope (Id = "Technical.NetworkA.SensorNodes",  
17 Strategy = "Dynamic", Priority = "High") {  
18 Policy (Id = "Restricted−Key-Length") {  
19 Rule ( Key-Length = "256-bit", Transport = "GPRS" )  
20 Policy (Id = "Quality−of−Protection") {  
21 Rule ( Key-Length = "128-bit", Battery = "Low" )  
22 }  
23 }  
24 }  
25 }  
26 }
```

4.3 Aspects

In order to disentangle crosscutting concerns in modern OO programming languages, aspect-oriented approaches have gained attention. Aspects provide designated means for encapsulating (non-functional) crosscutting concerns. Transferring aspect-orientation, including its concepts of pointcuts and advice, to policy languages offers users and developers a convenient and familiar solution for dealing with tangled and scattered assertions in policies.

Listing 8 presents a way of disentangling the scattered policy criticized in Section 2.3. Each pointcut defines a pattern over rules and assertions where the advice defines what assertions it adds to the matching rules. Introducing aspects explicates statements and bindings, as well as distinguishing functional from non-functional segments of a policy. Furthermore, the presented solution can avoid ambiguities using explicit execution order via priorities (e.g., line 24), which enables a better composable ability of different policy fragments.

Aside from integrating easily accessible constructs familiar from AOP, policies can also be empowered through user-defined functions, such as the NOT (ME) statement presented in the above listing. Here, NOT negates the predicate ME, ME refers to a provider that defined the above policy – thus the expression NOT (ME) matches all services not operated by the provider.

4.4 Multi-Paradigm Interpretation

Empowering policies with the interpretation of multiple paradigms allows them to incorporate different formalisms covering possibly orthogonal system views. It also gives users and developers more freedom to specify facets of a policy in the constructs they deem most appropriate—regarding, for instance, usability, understandability or brevity—for covering all pertinent facts of the view covered.

The preceding policy fragment in listing Listing 9 illustrates the integration of different policy constructs, i.e., a Finite State Machine and a Ponder role-based access control statement, into a generic WSPL policy. This permits access to primitives from policy languages that were designed to specify a specific system functionality within a joint generic context. What is special in such a com-
6. ACKNOWLEDGMENTS

The work presented in this paper was performed in the context of the Software-Cluster project EMERGENT (software-cluster.org). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. “01IC10501” and by the Center for Advanced Security Research Darmstadt (CASED, www.cased.de).

7. REFERENCES