HiPoLDS: A Hierarchical Security Policy Language for Distributed Systems

Matteo Dell'Amico[a], Gabriel Serme[b], Muhammad Sabir Idrees[a], Anderson Santana de Oliveira[b], Yves Roudier[a]

[a]*Eurecom, Sophia-Antipolis, France*
[b]*SAP Research, Sophia-Antipolis, France*

Abstract

Expressing security policies to govern distributed systems is a complex and error-prone task. Policies are hard to understand, often expressed with unfriendly syntax, making it difficult for security administrators and for business analysts to create intelligible specifications. We introduce the Hierarchical Policy Language for Distributed Systems (HiPoLDS), which has been designed to enable the specification of security policies in distributed systems in a concise, readable, and extensible way. HiPoLDS design focuses on decentralized execution environments under the control of multiple stakeholders. It represents policy enforcement through the use of distributed reference monitors, which control the flow of information between services. HiPoLDS allows the definition of both *abstract* and *concrete* policies, expressing respectively high-level properties required and concrete implementation details to be ultimately introduced into the service implementation.

Keywords:
security policies, service-oriented architectures, distributed systems

1. Introduction

Service-oriented architectures (SOAs) are a major software development pattern that builds applications based on loosely coupled services which can be run by different entities. Because of their complexity and of the varying degrees of trust between locations in which code is deployed and executed, it is challenging to make these systems secure. In particular, security is a *crosscutting* requirement: security-related code is generally scattered over several pieces of code and locations. What is worse, a local vulnerability or a mismatch between the security mechanisms adopted at different locations can have dire consequences, potentially putting large systems at stake.

The CESSA project[1] focuses on the daunting task of making large SOAs secure by using aspect-oriented structuring and modularizing security across administrative and technological domains. It is with this goal in mind that we introduce HiPoLDS (Hierarchical Policy Language for Distributed Systems). This language aims at being an efficient tool to express policies in diverse and complex distributed systems, where several entities interact in complex scenarios. SOAs are our motivating use case and they have driven our design, but HiPoLDS has been designed to be applicable to any kind of distributed system.

HiPoLDS provides the following features which are not present together in existing security policy languages:

- Allowing to describe the security policy also by way of *abstract* requirements: this should allow the writer of the policy to mention *which* security properties they want (*e.g.*, confidentiality, authentication, *etc.*) along with how they are implemented (*e.g.*, encryption, signatures. *etc.*).
- Expressing the security policy of a service-oriented architecture centrally despite its decentralized enforcement. A single abstract requirement (*e.g.*, confidentiality or authentication) often needs to be implemented distributedly with several pieces of code at different locations (*e.g.* encrypt somewhere and decrypt in another place, add security metadata at a sender and verify them at a proxy, and so on). HiPoLDS aims to make the relationship between the abstract requirement and its distributed implementation more obvious and modular.
- Keeping specifications clear and understandable, minimizing the need for code duplication and helping maintainability – even when policies are drafted cooperatively by several entities.

After discussing the state of the art on distributed policy languages in Section 2, we introduce the main constructs of our own language in Section 3. HiPoLDS is based on a hierarchy of *policy domains*, each of them being a set of locations where security policies apply. HiPoLDS security rules are handled by reference monitors (RMs) running at each policy domain and controlling the flow of information crossing their borders.

HiPoLDS has been designed by taking into account concrete practical use cases and deriving the features that were needed in such situations [1, 2]. We show in Section 4 various use cases highlighting how it makes it easier to express complex security requirements and how to refine specifications into security mechanisms. We discuss our plans towards a complete HiPoLDS implementation in Section 5, together

[1]`http://cessa.gforge.inria.fr`

with automated and semi-automated strategies to relate requirements with mechanisms that implement them. We finally conclude (Section 6) by highlighting open research issues related to HiPoLDS and outlining our agenda for future research.

2. Related Work

The expression of a security policy is central when it comes to describing how to secure a system. We review in this section a few of these approaches and in particular how appropriate they are for the distributed deployment of a service-oriented architecture. A large number of security policy specifications aim at mediating and restricting access to a central database. Those approaches cannot qualify for SOAs due to their distributed nature. In addition, the security policy of a SOA has to capture responsibilities about the enforcement of the security policy and the fact that not all execution environments where services are running can be controlled.

Even in decentralized settings, access control policies have generally been well covered. The SecPal language [3] is one such proposal for describing decentralized access control which formalizes the use of SPKI certificates. It is interesting in that, similarly to the underlying PKI infrastructure, it captures rather well the notion of trusted authorities and their respective competences for authentication. However, SecPal expressions are restricted to the access control model to be enforced and cannot describe any manipulation of messages required for more complex security policies. This means it also would not be very appropriate for analyzing complex and extensible protocols like those encountered in service-oriented architectures. Contrary to SecPal, some policy languages aim at extensibility rather than formal verifiability; this is the case of Li *et al.*'s approach [4]. The policy model is captured through facts and inference rules, which may be interesting for introducing additional concerns.

Information centric approaches like the Decentralized Label Model [5] or its variants aim at specifying formally the security properties of the information flow in a system. This description is implemented through the typing of information flows with labels, in particular confidentiality labels in the case of Myers and Liskov's label model. One advantage of this approach is that it is also very declarative and may describe many different properties beyond access control. Implementing policy enforcement may however be rather difficult to implement: some automation is required when moving to low-level operations, in particular with respect to the selection of the cryptographic mechanisms used and to the key distribution operations. All of those are left outside the security policy specification, thus likely preventing the customized combination of multiple encryption techniques; it is also implicitly assumed that this implementation will be "correctly" deployed, whatever that may mean to the security expert. Furthermore, while they are simple because of their high-level of abstraction, information flow security models require handling the declassification of information, which more or less breaks the regularity of the policy. Still, the Decentralized Label Model is at an advantage here compared with similar models by making this declassification operation explicitly described in the language.

It is worth noting that the security policy specifications described above approach the expression of the policy as a high level statement of security objectives or properties for the sake of separating the policy model from its implementation. However, by not considering low-level concerns related to policy enforcement they also fail to capture network boundaries, network domains, and the protocols between them, all which are however extremely important for the specification of relevant policies.

In contrast, the SPL language [6, 7] is quite inspirational in that it expresses the distributed enforcement of obligation policies at different levels of abstraction. Those policies easily map to reference monitors for enforcement. SPL also aims at providing a unifying framework for policies expressed at multiple places in a company. Still, SPL assumes that the enforcement is performed by a trusted entity which is not adapted for addressing SOA security in general. Furthermore, the policies expressed in SPL are simply access control related in that some information is authorized or forbidden, the expression of that requirement being essentially focused on the description of the reference monitor operation.

Ponder [8] is another language based on obligation and filtering whose expressivity is more extended. It too fails to express the existence of multiple entities for enforcement.

The Law Governed Interactions approach [9] (LGI) also constitutes a very interesting attempt at specifying policies over multiple domains, like network domains. LGI aims at rather diverse types of policies, even beyond security ones, encompassing for instance quality of service concerns. Policy enforcement in LGI is based on the realization of a policy based middleware in which communication is mediated by reference monitors between domains. In this approach, domains can be considered as governed by a mandatory policy, their law. However, the approach fails short to account for multiple stakeholders by not considering that the enforcement might not always be possible - or at least not by an authority that is trusted enough to ensure the application of the law. Unlike LGI, in HiPoLDS reference monitor do not need to be trusted by all participating entities, and need to be as trusted as the applications running in their domains.

All three approaches above feature the idea that the concept of domains is not only central to enforcement by an associated reference monitor but also central to the very specification of security policies. Our work builds on the idea that a domain does not only mean a consistent policy is enforced, but that the enforcement is under the control of a single authority. Given that authority model, this architecture may to some extent also help solve policy composition issues [10], even though we do not address this issue in this paper.

Using AOP for policy enforcement is not an original idea in itself. Several works apply AOP for building inline reference monitors for access control, such as [11, 12, 13] or to specify other requirements such as availability [14]. The particularity in our approach is its application to the information flow in a distributed systems, which requires the use of specific aspect mechanisms [15].

Hierarchical policies also require another approach to policy composition and conflict resolution. In previous work, conflicts are mostly solved by disambiguating among diverging policy decisions [16, 17, 18]. We advocate that the hierarchical organization of HiPoLDS policies essentially impacts the enabled information flows between domains controlled by different authorities. Handling such issues rather requires advanced negotiation techniques when policies cross domains.

3. Language Overview

In service oriented architectures, complex processes are carried out by several interacting entities. It is essential to support a concise way of specifying high-level policies that need to be applied in the whole architecture or in large parts of it, as well as fine-grained requirements that need to be applied in smaller domains. In HiPoLDS, we do this by defining the whole system architecture as a hierarchical structure of *policy domains*, and using *reference monitors* that enforce security policies at the border of policy domains.

A HiPoLDS document is composed of two parts: a declaration of the domain hierarchy, and a set of rules. The domain hierarchy, complemented by domain attributes, describes the global system architecture, while rules describe security policies that will take place. In this Section, we will introduce them along with the syntax, by means of simple examples.

3.1. Policy Domains

Policy domains can represent very different entities, such as corporations, individuals, down to the level of real or virtual machines, or even applications running on those machines. They are the scope for HiPoLDS rules, and they correspond to sets of entities that have particular properties that can be relevant to security (for example, a security domain can correspond to an organization, an individual, a place, a real machine, a virtual machine, and even a single application). Leaves of a policy domain hierarchy should be small enough that no security enforcement is needed for communications within a single policy domain. This is illustrated in Figure 1 on the following page, which shows an example hierarchy for a loan negotiation scenario, where a bank communicates with a governmental information system in order to evaluate the eligibility of its customers to loan aids. We will use this scenario as a running example throughout this paper.[2]

Policies regulate the exchange of information between different domains. They are enforced and monitored *at the borders* of policy domains. In this way it is possible to make sure that, for example, particular information sets remain confined within a domain (*e.g.*, to fulfill a privacy requirement) or get annotated with additional security metadata (*e.g.*, a Message Authentication Code or a cryptographic signature). The hierarchical structuring of policy domains allows the coexistence of policies that cover a wide set of locations (such as an organization) and of very specific ones (for example, access to a particular service).

Any domain can contain any number of subdomains, up to an arbitrary level of nesting. For a domain included within a parent domain, policies both in the enclosing and the inner domains will apply. This allows us to naturally define rules both at large (*e.g.*, organizations) and small (*e.g.*, services) scales. The hierarchy of policy domains also allows drafting a security policy cooperatively: the language supports the definition of rules that only apply within one, or more, policy domains. In our framework, administrators from involved entities should agree on “top-level” security policies applying to all domains, but they can be free to define additional security policies applying only to a domain of their competence. Data fields can be attached to policy domains, for example to contain encryption keys used by a principal.

In very complex scenarios, the number of policy domains might become very large, making the policy domain hierarchy unwieldy to handle manually. Currently, HiPoLDS does not handle this problem, which can be however managed by resorting to an external macro language such as, for example, GNU M4.[3]

In a deployed implementation, the definition of the domain hierarchy should also include a way to match the addressing scheme used for communication between services with the policy domains, in order to allow reference monitors to evaluate the origin and destination policy domain for a message.

Domain Attributes

We allow specifying an arbitrary number of additional *attributes* for each policy domain, when defining the policy domain hierarchy. Domain attributes are text labels allowing to attach additional information to the containment relationships implied by the hierarchical domain policy structure.

Domain attributes allow to specify policies that apply to several policy domains without the need to list them explicitly, allowing them to “cross” the policy domain hierarchy. For example, all policy domains corresponding to mobile devices can be labeled as “mobile”; afterwards, it will be easy to define policies that apply to all mobile devices in the policy domain hierarchy, or some sub-hierarchy of it, by writing policies that apply only to domains with a “mobile” attribute. Such a feature is essential for avoiding repetitions and keeping rules as terse as possible.

Domain attributes can be used for several purposes, such as describing what a policy domain corresponds to (*e.g.*, an organization, an individual or a device) or some technical architecture details (*e.g.*, the kind of operating system running on a machine).

All such information are potentially relevant with respect to the security policies, and domain attributes can be used to specify policies that apply to domains with common characteristics, even in different parts of the domain hierarchy. Using policy domain attributes increases maintainability: when a new policy domain is created, it will be sufficient to label it with the appropriate attributes and the relevant security policies will be applied to it as well.

[2]A complete HiPoLDS specification for this scenario is available in [1].

[3]`http://www.gnu.org/s/m4/`

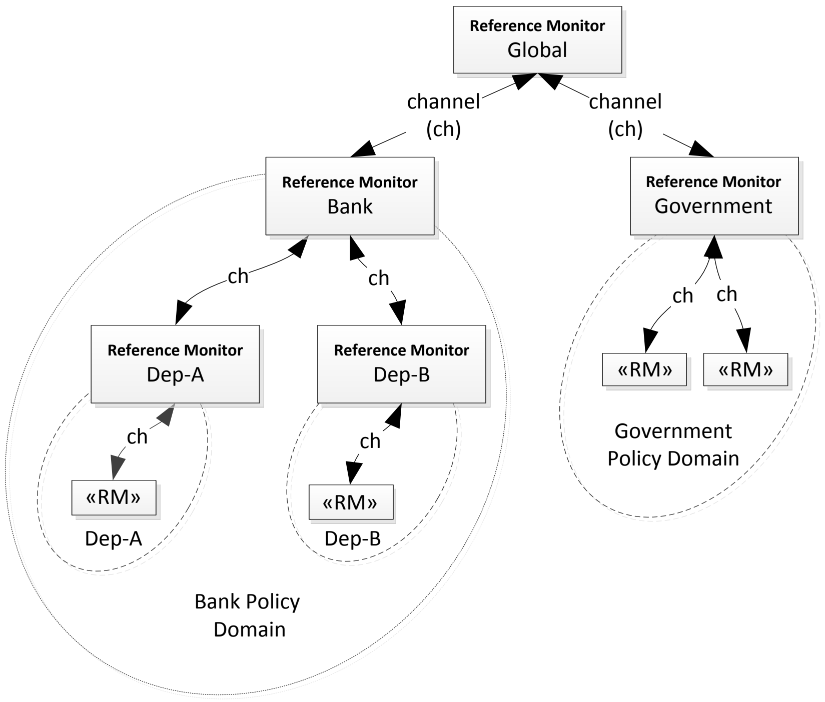

Figure 1: Domain Hierarchy Example

Example

We show the HiPoLDS declaration of part of the domain hierarchy for the loan negotiation scenario. The nesting of policy domains is expressed by enclosing inner domains in curly braces, and for each domain the list of domain attributes is written as comma-separated between parentheses. Here, domain attributes make it possible to differentiate the clerks from the manager. For brevity, we omit data items attached to domains, such as for example the encryption key IDs used by the bank employees.

```
Bank (organization) {
    Dept-A (Department, organization)
    {
        employee (manager, Department),
        Sub-department (sub-Domain, Department),
        {...},
    }

    Dept-B (Department, organization)
    {
        employee (clerk, Department),
        ...
    }
}
```

3.2. Reference Monitors

In our model, security is at stake if proper measures are not taken when information traverses policy domains. For example, due to a confidentiality requirement, one or more pieces of information should not be readable outside a given set of policy domains, and this requirement is not fulfilled when the information is sent unencrypted outside of the allowed domains, or the encryption keys are divulged. In our system model, a *reference monitor* per domain monitors all the information entering or exiting the domain and alters it as needed. A reference monitor communicates exclusively with the services in its own domain and with the reference monitors of the neighbors in the domain hierarchy (*i.e.*, parent and child domains).

Reference monitors are as trusted as anything in their policy domain: rather than being a trusted infrastructure (as, for example, in LGI), they are simply used to enforce and/or monitor the security mechanisms, separating them when possible from the business logic of the application. Reference monitors intercept, and take action, on communications across trust domain boundaries. They work similarly to “customs control”, enforcing restrictions about what gets in and out of a domain. Some actions that reference monitors can apply are:

- filtering: reference monitors can implement access control to resources outside of their original policy domain[4] by filtering unauthorized messages;
- cryptography: information can be encrypted, decrypted or signed when leaving or entering policy domains;
- managing security metadata: in our system, information is augmented with metadata that we label as *information tags* (see Section 3.4).

[4]Policy domains can be made as small as required; for example, to enforce access control to a service from any other location, a policy domain can enclose only the original service.

In other cases, reference monitors can enforce security policies by triggering actions that will take place in their policy domain.

3.3. HiPoLDS Rules

Rules are the way in which security requirements are specified in HiPoLDS. The form of a rule is `SCOPE {LEFTPART` $\rightarrow$ `RIGHTPART}`.

- The **scope** identifies the part(s) of the policy domain hierarchy in which the rule needs to be enforced. If omitted, the default scope for a rule is the whole domain hierarchy.
- The **left part** is a set of comma-separated clauses that describe the conditions that trigger rule enforcement. The rule is enforced when all the clauses on the left part are true.
- The **right part** describes the properties that are required to hold. The rule is satisfied when all the clauses on the right part are true.

A first example of a rule is the following one:

$$\texttt{x} \rightarrow \texttt{x}\ is\ \texttt{confidential}(Bank, Government)$$

The scope here is omitted, meaning that the rule applies to the whole policy domain hierarchy. The left part of the rule, in this case, matches the only variable x. In HiPoLDS, variables match pieces of information; if they appear on the left side, they are implicitly quantified universally. In this case, the variable x therefore matches any piece of information exchanged in the whole domain hierarchy. If a variable appears only on the right side of the rule, it is instead implicitly quantified existentially, meaning that security enforcement mechanism must ensure that such an assignment to the variable exists such that the right part of the rule is satisfied.

On the right side, *confidential* is a security property – specified by a list of domains – that must be ensured. Security properties in HiPoLDS are preceded by the **is** keyword. The *confidential* property requires that pieces of information will not be readable outside of the specified policy domains – Bank and Government, in this case. Security properties can also describe constraints on the content of information tags or on the composition of information tag sets. In that case, the usual set operators are then used.

Abstract and Concrete. The rule described above requires that the *confidential* property is ensured, yet it can be described as underspecified in the sense that it can be implemented in different ways. For example, all messages that leave Bank or Government can be encrypted with keys only available in those domains, or messages can just be filtered when they leave any of those domains. We refer to rules that can be implemented in several possible ways as *abstract* rules, as opposed to *concrete* rules that give a complete specification that can be executed by reference monitors. Since abstract rules are less verbose and more focused on the security properties that are needed, we consider the ability to express them as very beneficial towards having clear and maintainable security policies. The development of inference techniques that would help writers of security policies derive concrete rules starting from high-level, abstract ones is currently an open issue on which we are still working.

Composition. Rules are *not monotonic*. For example, consider a sub-domain B of domain A. If we consider requirements on data confidentiality, a piece of information can be allowed to be readable only within B and not in the rest of A, but also the opposite can apply: if B for some reason is considered "less trusted" than the rest of A (*e.g.*, a mobile device that can fall more easily in the hands of an attacker), then restrictive rules can be applied whenever some data is sent to B.

It is possible that rules will require actions that are impossible to satisfy or in conflict with other rules. We plan to investigate how to detect conflicting rules, both statically (*i.e.*, when drafting the security policy) and at runtime, and on determining ways to manage them. Conflicts within a policy domain are the easiest to solve as they only correspond to local policies as defined by the same authority.

In the current proposal, we limit ourselves to an order-based prioritization of rules within a policy domain, in the style of most firewall policies; other approaches for solving conflicts have been vastly explored in the literature and might be applied as well. In contrast, conflicts between policies defined in different domains are harder to solve as they are defined by potentially different authorities and thus require some negotiation. Due to the style of HiPoLDS policies which only adds further security constraints to the diffusion of information flows between policy domains, those conflicts cannot increase the rights granted to principals; instead they may impede communication between two policy domains, especially if an intermediate domain prohibits information to flow across. In this current proposal, we will limit ourselves to a simple priority rule, by choosing the most specific rules (*i.e.*, those defined for inner policy domains) and, to discriminate between rules defined at the same hierarchy level, we will give priority to the one defined first.

3.4. Information Tags

Information tags are free-form text labels representing some security metadata attached to information and that categorizes it. It is possible to define HiPoLDS rules that apply to information that has particular tags; when combined with domain attributes this allows us to naturally define policies that apply to large sets of information and span different policy domains. Reference monitors manage information tags and use information tags to decide which actions to take. For example, based on the tags it has, information can be filtered, transformed (*e.g.*, through encryption, stripping of confidential information, sanitization against injection attacks) and/or rerouted. Information tags are stripped before sending information to the original services, which will behave as if no security mechanisms were put in place.

3.5. Hierarchy vs. Attributes

HiPoLDS provides two different ways to express security policies that apply to several different policy domains: cluster them in the policy domain hierarchy, or adopt domain attributes to express a logical grouping. The difference between the two is that an outer policy domain grouping various inner domains should be able to host a reference monitor that filters all traffic for inner domains. This solution is efficient when the network topology actually matches with the policy domain hierarchy; such a solution can allow for more efficient enforcement (for example, a confidentiality requirement within a certain organization can be enforced by analyzing only traffic leaving that organization). Mismatches can be solved by using virtualized network topologies such as VPNs (Virtual Private Networks).

The hierarchical grouping, however, can become inefficient when traffic has to be rerouted through "parent" reference monitors several times; consider the case of portable and mobile devices and the phenomenon of "BYOD" (Bring Your Own Device). In such cases, using virtual network channels to route traffic might be inefficient – leading to a multiplication of traffic passing through reference monitors, possibly introducing bandwidth bottlenecks and increasing costs. Enforcement of policies can be better handled right at a reference monitor at the device level. In that case, the proper way to group domains with similar policies is via domain attributes.

4. Examples

After introducing the basic structure of HiPoLDS, we now show how its features can be adopted in a realistic case. We illustrate the relationship between concrete and abstract rules, discuss the role of reference monitors, and show how domain attributes can be used to describe role-based security rules.

4.1. Stateless Policies

In this section, we first provide some examples of simple rules that can be executed independently of any other context to showcase the syntax of HiPoLDS.

Origin Authentication. The following example illustrates how the authentication of the origin of a message can be defined in the policy using a digital signature mechanism. The verification of the signature will be enforced by the reference monitor. The following example also illustrates how an information tag can be used in HiPoLDS to work on the contents of a message:

$$\begin{array}{l} \texttt{Bank}\,\{ \\ \quad \texttt{m} : \textit{message},\ \ \texttt{x} : \textit{customer-info} \in \texttt{m}.\textit{contents}, \\ \quad \texttt{m}.\textit{from} == \texttt{y} :: \textit{employee},\ \ \texttt{y}.\textit{public\_key} == P_k \\ \quad \to \texttt{x}\ \textit{is}\ \texttt{signed}(P_k) \\ \} \end{array} \tag{1}$$

In this example, the scope of the rule is the `Bank` domain. In its place, a domain attribute could have been there (for example, `bank`) to specify that the rule applies to all banks in the domain hierarchy. In this rule, we see for the first time the '`:`' and '`::`' constructs, which are used respectively to match variables representing data with information tags and those representing domains with domain attributes. In addition, data fields on information and on policy domains are accessed via the dot notation seen in `m.contents` and `m.from`.

The left side of the rule uses information tags and domain attributes to match any message m sent from an employee y in the Bank. Since variables appearing on the left side of the rule are quantified universally, the contents of the message m are bound to x. Then y's public key is bound to the variable P_k; finally, the right side of the rule requires that the message is signed with the key P_k. When the message is sent, reference monitors will verify the state of the message; if the left side matches and the right side does not (*i.e.*, the message is not signed) the appropriate reference monitor has to process the messages so that the right part of the rule is verified. In this example, this can happen in two ways: if the reference monitor has access to the private key matching with P_k, it can add a signature to the message. Otherwise, the other possibility is dropping the message.

In summary, the rule above can be read as follows in plain English: *"The following rule applies only to the policy domain of Bank. For each message m sent by an employee y with a public key P_k, P_k must be used to sign the contents of the message."*

Four Eyes Principle. A message may need to be processed by two different principals before it is deemed acceptable. It is possible to specify such a rule in HiPoLDS by combining two origin authentications. The location of this rule in the set of policy domains will determine where in the overall architecture such a check will be done. In the following example, we are just checking that a financial operation has been checked both by the front-office and by the back-office. All attributes might be used to characterize the two principals involved:

$$\begin{array}{l} \texttt{m} : \textit{accept\_loan}, \\ \quad \texttt{e1} : \textit{front-office} \in \texttt{m}.\textit{contents}, \\ \quad \texttt{e2} : \textit{back-office} \in \texttt{m}.\textit{contents}, \\ \quad \texttt{e1.key} == P_{k1}, \\ \quad \texttt{e2.key} == P_{k2}, \\ \quad \to m\ \textit{is}\ \texttt{signed}(P_{k1}), \\ \quad P_{k1}\ \neq\ P_{k2} \end{array} \tag{2}$$

Separation of Duty. Separation of duty will essentially be addressed based on the contents of messages. The policy must specify which part of a message cannot originate from one principal. Such specifications can of course be combined with roles. The following policy describes that a message can be forwarded only if two different people originated the fields `customer-info` and `customer-account-value`, as proven by their respective signatures:

$$\begin{aligned}
&\texttt{m} : \textit{message}, \\
&\ \texttt{x} : \textit{customer-info}\ \epsilon\ \texttt{m}.\textit{contents}, \\
&\ \texttt{y} : \textit{customer-account-value}\ \epsilon\ \texttt{m}.\textit{contents}, \\
&\ \texttt{x.from} == \textit{e1::employee},\ \texttt{e1.key} == P_{k1}, \\
&\ \texttt{y.from} == \textit{e2::employee},\ \texttt{e2.key} == P_{k2}, \\
&\ \rightarrow x\ \textit{is}\ \texttt{signed}(P_{k1}), \\
&\ y\ \textit{is}\ \texttt{signed}(P_{k2}), \\
&\ P_{k1} \neq P_{k2}, \\
&\ \textit{not}(y\ \textit{is}\ \texttt{signed}(P_{k1})), \\
&\ \textit{not}(x\ \textit{is}\ \texttt{signed}(P_{k2}))
\end{aligned} \tag{3}$$

4.2. Stateful Policies

In this section, we provide use cases featuring stateful policies. State handling is generally necessary to define more involved security properties and to define history based security mechanisms.

Counting Messages. In practice, accountability objectives for instance generally involve logging the traffic that has been observed. This is often implemented by modifying the state of the reference monitor. This concept can be illustrated in the case where we want to track how many messages have been sent by employees in a company. This particular monitoring policy can be described using a counter variable attached to every employee, and whose value is incremented after the message tracked is observed, as follows:

$$\begin{aligned}
&\texttt{m} : \textit{message},\ \texttt{m}.\textit{from}\ ==\ \textit{y::employee}, \\
&\ \texttt{y}.\textit{msg\_count}\ ==\ z \\
&\ \rightarrow \texttt{y.msg\_count}\ ==\ \texttt{z+1}
\end{aligned} \tag{4}$$

Chinese Wall. Conflicts of interests are typically avoided using the Chinese Wall security policy [19]. This is also a typical example of the need for the expression of a stateful security policy, since one must remember the accesses requested and obtained by principals. The reference monitor typically needs to update its logs based on the data that have been sent and will filter accesses declared illegal after the policy based on the contents of these logs. Again, this can be implemented using a state variable attached to the principal. This variable stores the set of contents accessed by the message recipient so far. This set is compared with conflict of interest classes, as defined by Brewer and Nash, which themselves have to be first statically initialized based on information tags about all possible data contents of messages. The following dynamic check describes in HiPoLDS how attributes that should not be sent to the recipient can be filtered out (without dropping the entire message):

$$\begin{aligned}
&\texttt{m} : \textit{message},\ \texttt{x}\ \epsilon\ \texttt{m}.\textit{contents} \\
&\ \rightarrow \texttt{m}.\textit{contents}\ \texttt{-=}\ (\texttt{e}.\textit{content\_accessed} \cap \texttt{x}.\textit{conflict\_set}), \\
&\quad \texttt{e}.\textit{content\_accessed}\ \texttt{+=}\ \texttt{x}
\end{aligned} \tag{5}$$

Given the universal quantification on x, this rule actually might result in the addition of several elements to the `content_accessed` set and the removal of several message elements due to conflicts of interest.

4.3. Abstract and Concrete Rules

Let us consider a security requirement of the following form: *"Customers' private information should only be disclosed to the Bank and the Government, and its integrity has to be guaranteed"*. Such a requirement would translate to the following HiPoLDS *abstract* rule:

$$\begin{aligned}
&\texttt{m} : \textit{message},\ \texttt{x} : \textit{customer-info}\ \epsilon\ \texttt{m}.\textit{contents} \\
&\ \rightarrow \texttt{x}\ \textit{is}\ \texttt{confidential}(\textit{Bank}, \textit{Government}), \\
&\ \texttt{m}\ \textit{is}\ \texttt{integrity\_verified}
\end{aligned} \tag{6}$$

In this case, we use the `customer-info` information tag to denote messages that contain the kind of information that is affected by the rule, and limit the disclosure of data to the Bank and Government domains, and to their sub-domains. In this case, the abstract properties we require are `confidential` and `integrity_verified`.

Asymmetric Encryption. Since this is an abstract rule, it can be implemented in several ways. A first option is adopting asymmetric cryptography: for example, when a message is sent from the Bank to the Government, the following rule might be applied in the reference monitors in the Bank domain:

$$\begin{aligned}
&\texttt{Bank}\ \{ \\
&\quad \texttt{m} : \textit{message},\ \texttt{x} : \textit{customer-info}, \\
&\quad \texttt{m}.\textit{to}\ ==\ \texttt{t}\ \textit{in Government} \\
&\quad\ \rightarrow \texttt{x}\ \textit{is}\ \texttt{asym\_encrypted}(t.P_k) \\
&\}
\end{aligned} \tag{7}$$

In this case, `asym_encrypted` is a *concrete* rule applying to all the messages that are sent to any recipient in the Government domain (*i.e.*, whose `to` field is within a policy domain contained in `Government`). This is a concrete rule because it dictates the specific mechanism to use in order to obtain the required property, which is confidentiality.

Such a rule has to be accompanied by other rules: the companion rule enforcing decryption when messages are received in the Government, and a set of analogous rules for messages sent from the Government to the Bank which are tagged with `customer-info` as well.

Symmetric Encryption. Other concrete implementations of the same abstract requirement are possible. For example, this can be done with a symmetric cryptography implementation using a shared key:

$$\begin{array}{l}\texttt{Bank}\ \{\\ \quad \texttt{m} : \textit{message},\ \texttt{x} : \textit{customer-info},\ \texttt{m}.\textit{to} == \texttt{t}\ \textit{in Government}\\ \quad\ \rightarrow \texttt{x}\ \textit{is}\ \texttt{sym\_encrypted}(\textit{Bank.shared\_key})\\ \}\end{array} \tag{8}$$

The above concrete policy rule can implement the required abstract property; however, there are cases for which such a solution would not be acceptable: for example, if the abstract property of non-repudiability were requested, it would not be achievable with only this mechanism.

Multi-Step Encryption. More elaborate scenarios are conceivable: for example, if a reference monitor (say, on a mobile device representing a subdomain of Bank) is not considered to be trusted enough to hold a system-wide shared key - like in the example before - and not powerful enough to process asymmetric encryption, multi-step protocols can be envisaged. In this case, for example, the reference monitor on the mobile device can use a shared key to use symmetric encryption with the Bank reference monitor, which can then re-encrypt the messages towards the intended recipient with asymmetric encryption. Such a policy is within the expressive capabilities of HiPoLDS, and can be expressed as follows.

$$\begin{array}{l}\texttt{MobileDevice}\ \{\\ \quad \texttt{m} : \textit{message},\ \texttt{x} : \textit{customer-info},\\ \quad \texttt{m}.\textit{to} == \texttt{t}\ \textit{in Government}\\ \quad\ \rightarrow \texttt{x}\ \textit{is}\ \texttt{sym\_encrypted}(\textit{MobileDevice.shared\_key}),\\ m : step1\_applied\\ \}\\ \texttt{Bank}\ \{\\ \quad \texttt{m} : step1\_applied\\ \ \rightarrow \texttt{m}\ \textit{is}\ \texttt{sym\_decrypted}(\textit{MobileDevice.shared\_key}),\\ \quad \texttt{m}\ \textit{is}\ \texttt{asym\_encrypted}(m.to.P_k),\\ \}\end{array} \tag{9}$$

In this case, the step1_applied information tag is used to mark the first processing step where it is applied; processing will further continue at the Bank reference monitor. As before, further matching rules will decrypt messages at the recipient, and deal with sending messages in the opposite direction.

We consider the ability of expressing both abstract and concrete rules as a key feature of HiPoLDS; in Section 5.1 we discuss our plans for deriving or verifying concrete policies based on abstract ones.

4.4. Roles and Policy Domains

It is worth noting that rules based on roles can be expressed via HiPoLDS. Indeed, roles can be expressed by assigning policy domain attributes to policy domains that represent individuals. The following (abstract) rule states that all messages tagged as classified should remain confidential between managers:

$$\texttt{m} : \textit{classified} \rightarrow \texttt{m}\ \textit{is}\ \texttt{confidential}(\textit{manager})$$

In this case, we remind that manager is a domain attribute, and this rule would be equivalent to enumerating all policy domains with the manager attribute. Using domain attributes in this way helps maintainability and avoids repeating the same rule for different domains. The rule can be implemented using concrete rules similar to what we have seen in Section 4.3.

Let us furthermore suppose that we want to avoid sending classified messages to mobile devices, even if they are owned by a manager (*i.e.*, they are subdomains of a domain with manager attribute). Such a rule writes as

$$\texttt{m} : \textit{classified} \rightarrow \texttt{m}\ \textit{is}\ \texttt{filtered}(\textit{mobile})$$

In this case, filtered is a new property requiring that messages should not arrive to the listed domains. Again, mobile is a domain attribute and using it is equivalent to listing all domains tagged as mobile.

To enforce this rule, reference monitors in a manager domain with mobile subdomains should enforce the filtering. This kind of rule is applicable because correctly-behaving reference monitors communicate with each other only through the hierarchical channels as shown in Figure 1, so the parent node is the only point from which information can reach a domain. It is exactly because, in this case, mobile domains are feared to not behave correctly (*e.g.*, have side communication channels) that confidential information is filtered before reaching them.

Here, we reinforce the fact that each reference monitor is as trusted as the domain it is in, and such trust is non-monotonic. In fact, the only reference monitors that handle confidential information unencrypted are those in the domains that have access to it. We point out that this might mean that reference monitors at high levels in the hierarchy might not have any concrete rule to apply – this means that they can effectively be removed. In particular, the top-level global reference monitor could be complex to deploy and to implement, and concrete rules that do not need it could be advisable.

5. From Specification To Enforcement

Until now, we focused on the design and expressive power of HiPoLDS. In this Section, we turn our attention towards the implementation of an enforcement architecture that implements HiPoLDS on real systems.

Globally, a HiPoLDS reference monitor works as an application-level gateway (ALG), that is, a firewall-like entity that is put between services and the rest of the network; its responsibility is to filter and mangle traffic according to HiPoLDS concrete rules. ALGs for web services have been studied by academia [20], [21], industry [22], and standard bodies [23]; commercial solutions [24, 25] are also available. Compared to these approaches, HiPoLDS provides a framework to configure reference monitors according to a set of more abstract overarching security policy, automating the application of matching policies in different reference monitors (*e.g.*, message encryption on the sender side is matched with decryption at the receiver).

Figure 2 displays the architecture of the four main components we design for the correct enforcement of rules, using an FMC [26] diagram:

1. *Policy Engine*, putting in relation abstract and concrete policies;
2. *Traffic Analysis*, relative to extracting information relevant to security from the traffic observed on the network;
3. *Decision Engine*, taking as input the output of traffic analysis, and putting them in relation to the concrete policies in order to decide how to operate on traffic.
4. *Enforcement Engine*, enacting the transformations on traffic dictated by the decision engines.

In the rest of this Section, we will describe the responsibilities and interactions of each component; we will also provide details on the implementation choices we have taken for the ongoing development of a prototype reference monitor.

5.1. From Abstract to Concrete Policies

The most realistic approach to derive concrete policies into contrete ones would be to rely on the expertise of the security administrator to define concrete policies from abstract ones. Expert knowledge about domain topology and its components is much more precise than fully automated processing, that would ignore details that can be fundamental for security (e.g. protecting password files). The process can be supported by tools to check the abstract and concrete policies, such as model checkers, which can work without much intervention. For instance, the ASLAN++ language and system [27] allows to model communication protocols for the verification of security goals.

The definition of a (semi-)automated refinement process is itself a research challenge, but feasible. It is necessary to reduce the semantic gap from abstract to concrete policies: a translation framework linking the abstract concepts of HiPoLDS policies to lower level system concepts. One of the tasks to achieve it is to identify decidable classes of HiPoLDS policies and to establish the correctness of the generated concrete policies, considering the formal semantics at the abstract and concrete levels. Remark that multiple correct concrete policies can be associated to one abstract policy. Another problem to be solved can be code optmization: the automated concrete policy generation could produce possible overhead. For instance, encrypting a communication channel more than once, as soon as the policies governing a domain hierarchy may require confidentiality under overlapping conditions.

Likely the best solution is the combination of human expertise and tools to generate concrete policies. This would reduce the number of assumptions about the domain in question to map abstract to concrete policies (a large number of semantic relationships among the tags, attributes, roles and the domain hierarchy). An automated tool would be able to produce a partial concrete policy as an output that would need to be manually edited or corrected by a security expert. As soon as basic considerations about enforcement could be reasonably handled by the automated refinement process, the effort of the security expert could be reduced.

5.2. Traffic Analysis

In order to enact the concrete policies, reference monitors need to intercept all communications that traverses policy domain boundaries. An appropriate choice for this scenario is to implement them as *application-level firewalls*, intercepting all traffic to or from their policy domain, and performing the appropriate filtering and/or mangling before distributing it to the policy domain. Traffic that reaches the analyzer will not propagate to the rest of the network until the enforcement engine – after performing all the due actions – dictates to do so.

A key part of reference monitors is *traffic analysis*: from all information traveling over the network, extracting the set of data that is relevant with respect to the choice of how to handle messages passed. This data will then be passed to decision engines, which are described in Section 5.3.

The type of information that a traffic analyzer might need to extract is dependent on the concrete rules defined on the reference monitor it is running on, and can include information coming from different layers of the network stack – e.g., IP address information and application-level payload. In addition, traffic analyzers will parse information tags sent by other reference monitors.

Information collected by the traffic analyzer will be presented to the decision engines in the form of additional information tags; as we have pointed out, HiPoLDS is meant to apply generically to distributed systems, and aims to be viable even for applications that are using heterogeneous ways of performing messages. This requires defining ad-hoc rules to extract information from the data payload; this is dependent on the details on the message exchange policy used, and is therefore outside the scope of this work.

We are currently working on a prototype implementation of a reference monitor that handles web services communicating, and uses the Wireshark network analyzer[5] to obtain information about data exchanged between Web services, analyzing protocols at the level of TCP/IP, HTTP requests and JSON payload. By using Wireshark, we also make it easy to analyze data at different layers of the networking stack, and allow for further extension of the reference monitor.

5.3. Decision Engines

The decision engines are responsible for applying the concrete rules on the information about traffic that is output by the analyzer. The output of decision engines is a set of actions to

[5]`http://www.wireshark.org`

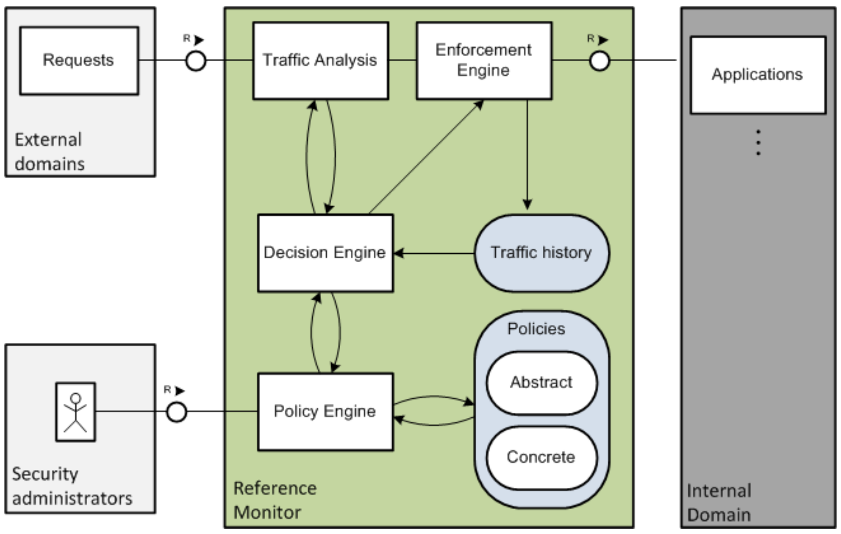

Figure 2: Enforcement Architecture

apply to data, which will afterwards be passed to the enforcement engines described in Section 5.5.3.

In our current development, we are implementing decision engines in Prolog. Traffic information coming from the analyzers is output as Prolog facts; furthermore, concrete rules are also represented as Prolog facts. In this way, we can make it possible to use Prolog's unification engine in order to associate traffic with maching rules. The output of decision engines is, again, a set of Prolog facts that represent all the actions that the reference monitor applies on the data.

Since the form of the two languages is rather similar, we consider that translation from HiPoLDS rules to Prolog rules is a reasonable task.

It is worth noting that it is easy to define sets of conflicting actions – such as, for example, "forward message *m* to user *x* in policy domain *y*" and "ensure that message *m* is not sent outside the current policy domain". Such a conflict detection is trivially done by creating additional Prolog rules. This solution will detect conflicts "at run-time", when they are generated. Detecting conflicts statically, preventing them to ever reach this stage, is preferable and it is a topic for further work. This simple solution, however, makes it possible to detect some problems by performing testing, and to alert security administrators in the case such problems happen on a "live" system.

5.4. Enforcement Engines

Enforcement engines are the reference monitor components that have the duty to put into action the directives that are output by the decision engines. They include all actions that may be dictated by the concrete rules, and imply not only message filtering, but also any kind of message alteration that makes sense from the point of view of security. For example, an enforcement engine should be able to add or remove security metadata such as signatures or message authentication codes, strip or encrypt confidential information, or decrypt it when it is the case. Ultimately, enforcement engines will have the task of sending altered traffic over the network.

We remind that the HiPoLDS philosophy is to separate as much as possible the security concerns, such that services should behave without taking them into account: they are already taken care of by reference monitors. This results in the fact that most of the security metadata will be stripped away before being sent to services; moreover, data meant to be accessed by the services will be unencrypted. A particular kind of alteration that will always be done relates to information tags: since they are meant to be exclusively handled by reference monitors, they will be stripped before being sent to services in the same policy domain.

Enforcement engines will receive input from decision engines – in order to decide which actions to take – and from traffic analyzers: to perform manipulations over traffic, and to ultimately send it, they will need to access the raw data of the traffic.

A particular action that enforcement engines apply is verification (*e.g.*, verification of signatures of message authentication codes): in this case, they will obtain an output that will be the base for further decisions by the decision engine: as such, the enforcement engine will send a new fact to the decision engine (*e.g.*, signature *s* is not verified); the decision engine will then generate new decisions based on this outcome that will be then applied by the enforcement engine, possibly requiring a further feedback between decision and enforcement engines.

5.5. Supported Properties

In our current implementation, we focused on a small yet representative set of security policies, covering three abstract properties and their concrete implementations: confidentiality, authentication and availability; they are described in the following. Table 1 on the next page lists the abstract and concrete properties currently implemented.

We assume that encryption keys are distributed beforehand to reference monitors; in our communication model all messages are exchanged on a one-to-one basis: there is no broadcast or multicast. We consider that each reference monitor holds an

Table 1: Supported security properties.

Abstract property	Concrete mechanism
Confidentiality	Drop End-to-end asymmetric End-to-end symmetric Boundary-to-boundary asymmetric Boundary-to-boundary symmetric
Authentication	Asymmetric signature MAC
Availability	Do-not-drop

asymmetric key pair, whose public part is known to all reference monitors; moreover, whenever symmetric communication between pairs of reference monitors is required, we assume that there is a shared symmetric encryption key between the two reference monitors.

5.5.1. Confidentiality

A *confidentiality* requirement specifies that messages matching the specification of the left-hand side of the HiPoLDS rule will not be readable outside a set of allowed domains, specified on the right hand side of the rule. This requirement can be satisfied by several strategies: the simplest one is the *drop* mechanism, whereby matching messages that should be sent to an unauthorized reference monitor are instead dropped. This, of course, can harm the functionality of the system: in Section 5.5.3 we show how availability requirements are modeled here.

When it is unacceptable to drop messages that would be routed through unauthorized domains, confidentiality is attainable through encryption. We consider concrete strategies that adopt both symmetric and asymmetric encryption; in addition, we consider two different kinds of encryption strategies, depending on the reference monitor where the encryption is carried out: *end-to-end* and *boundary-to-boundary*.

End-to-end encryption is the typical case where confidential messages are encrypted at the sender and decrypted at the receiver; this, however, may place unsustainable load on underpowered devices. Boundary-to-boundary, instead, involves lazily delegating encryption to other reference monitors that are at higher layers of the policy domain hierarchy. We remind that, due to the tree structure of the policy domain hierarchy, a message has to be sent through a sequence of relay nodes; since this hierarchy has a tree form, the path is unique. The boundary-to-boundary strategy employs this property: whenever a confidential message has to be routed through an untrusted reference monitor, the last trusted "boundary" node encrypts the message so that it will be unencrypted by the next trusted monitor that will have to handle it (*i.e.*, the other "boundary" of the untrusted zone).

This example shows how the same requirement can be implemented by way of different concrete policies, and using one or another depends on criteria such as the possible conflicts that may raise between policies, computational load and scalability issues. We are considering more elaborate policies that use domain attributes to identify the capabilities of each node, and choose the points of encryption and decription as the result of an optimization strategy.

5.5.2. Authentication

An authentication property requires that the sender of messages, and the message integrity, is verified at the receiver side. We consider two different concrete implementation for this: one using traditional symmetric/asymmetric key pairs, another one using a shared secret which is used to compute a Message Authentication Code using symmetric cryptography. Additionally, further options can be specified to support message signature and verification not only at the endpoints (source and destination), but also at reference monitors in the path between the two endpoints. This is done to ensure that messages have been delivered through the correct path, undergoing the necessary security checks.

5.5.3. Availability

In the context of HiPoLDS, an availability property specifies that the concrete mechanisms that are put in place do not harm the required system functionality: if the system does not implement its required functionality in the first place, it is not the job of the reference monitors to provide it.

The semantics of an availability requirement (specifying a set of domains where this requirement should be respected) is therefore raising conflicts whenever any concrete mechanism would "break" the availability requirement; currently, conflicts are detected at run-time as described in Section ; it is in our agenda to consider how to detect such conflicts statically.

Within the current list of implemented mechanisms (*cf.* Table 1), the *drop* mechanism can hurt availability requirements; therefore, a "do-not-drop" decision is taken by the decision engine when a message should remain available. The decision engine outputs both a "do-not-drop" and "drop" decision, an error is raised and communicated out-of-band to the security administrator.

6. Conclusion

In this paper, we introduced the design of and main implementation directions of a new security policy language, HiPoLDS, intended for specifying rules regarding the placement of security measures, access control, usage control, and similar concerns in complex distributed systems typically encountered in service-oriented architectures. The main idea behind HiPoLDS is to separate between the application logic and handling of security in order to mitigate the complexity of designing such systems, and to make them less subsceptible to errors. We designed the features of HiPoLDS based on various real-world use cases. As a result, we believe that this language is expressive enough to describe tersely a large amount of real-world policies.

Abstract policies also play the role of high-level security requirements to a certain extent, defining *what* should be done in order to guarantee security in the system; concrete policies instead specify *how* it should be done. We are starting with simple mechanisms to derive concrete policies from abstract ones, which brings up interesting open issue: the same abstract requirement can be implemented with several concrete strategies at once. One of these strategies must be selected based on several criteria: computational efficiency, simplicity, maintainability, and even resilience in case weaknesses are discovered in security protocols. In addition, we will investigate whether concrete policies can conflict, in which situations, and how to detect them before deploying and testing the system.

HiPoLDS has been inspired by service-oriented architectures; however, the way it has been specified make it agnostic to the particular architecture it is developed in. We therefore plan on developing different enforcement mechanisms so that it may be executed on complex hybrid architectures using different message passing architectures and deployment models.

A further topic to consider is scalability. It is interesting to analyze where concrete policies can be easily distributed and decentralized, in order to avoid possible performance bottlenecks and points that could be easily attackable by denial-of-service attacks. In conjunction with the former point about the very choice of concrete policies that implement a set of given abstract policies, this would make it possible to consider infrastructure for security that scale well. In addition, in order to avoid the overhead of sending data over the network for analysis by the reference monitors, we are also considering a different implementation of reference monitors, where they are "woven" into the application code using techniques from aspect-oriented programming. This solution has the potential of solving the performance overhead due to reference monitors.

References

[1] M. Dell'Amico, M. S. Idrees, Y. Roudier, A. S. de Oliveira, G. Serme, G. Harel, Language definition for security specifications, Deliverable D2.2, The CESSA project, 2011. http://cessa.gforge.inria.fr/lib/exe/fetch.php?media=publications:d2-2.pdf.

[2] R. Douence, H. Grall, I. Mejía, J.-C. Royer, M. Südhold, M. S. Idrees, Y. Roudier, J. Leroux, F. Rivard, J. Pazzaglia, G. Serme, Survey and requirements analysis, Deliverable D1.1, The CESSA project, 2010. http://cessa.gforge.inria.fr/lib/exe/fetch.php?media=publications:d1-1.pdf.

[3] M. Y. Becker, C. Fournet, A. D. Gordon, SecPAL: Design and semantics of a decentralized authorization language, J. of Computer Security 18 (2010) 619–665.

[4] J.-X. Li, B. Li, L. Li, T.-S. Che, A policy language for adaptive web services security framework, ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing 1 (2007) 261–266.

[5] A. C. Myers, B. Liskov, Protecting privacy using the decentralized label model, ACM Transactions on Software Engineering and Methodology 9 (2000) 410–442.

[6] C. Ribeiro, A. Zuquete, P. Ferreira, P. Guedes, SPL: An access control language for security policies with complex constraints, in: Proc. of NDSS.

[7] C. Ribeiro, P. Ferreira, A policy-oriented language for expressing security specifications, International Journal of Network Security 5 (2007) 299–316.

[8] N. Damianou, N. Dulay, E. Lupu, M. Sloman, The Ponder policy specification language, Policies for Distributed Systems and Networks (2001) 18–38.

[9] C. Serban, W. Zhang, N. Minsky, A decentralized mechanism for application level monitoring of distributed systems, in: Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on, IEEE, 2009, pp. 1–10.

[10] L. Bauer, J. Ligatti, D. Walker, A language and system for composing security policies, Technical Report TR-699-04, Princeton University, 2004.

[11] A. S. de Oliveira, E. K. Wang, C. Kirchner, H. Kirchner, Weaving rewrite-based access control policies, in: P. Ning, V. Atluri, V. D. Gligor, H. Mantel (Eds.), FMSE, ACM, 2007, pp. 71–80.

[12] E. Song, R. Reddy, R. B. France, I. Ray, G. Georg, R. Alexander, Verifiable composition of access control and application features., in: E. Ferrari, G.-J. Ahn (Eds.), SACMAT, ACM, 2005, pp. 120–129.

[13] D. S. Dantas, D. Walker, Harmless advice, in: J. G. Morrisett, S. L. P. Jones (Eds.), POPL, ACM, 2006, pp. 383–396.

[14] F. Cuppens, N. Cuppens-Boulahia, T. Ramard, Availability enforcement by obligations and aspects identification., in: Proc. ARES, pp. 229–239.

[15] M. S. Idrees, G. Serme, Y. Roudier, A. S. D. Oliveira, H. Graal, M. Sudholt, Evolving security requirements in multi-layered service-oriented-architectures, in: Proc. SETOP, pp. 190–205.

[16] P. A. Bonatti, S. D. C. di Vimercati, P. Samarati, An algebra for composing access control policies., ACM Trans. Inf. Syst. Secur. 5 (2002) 1–35.

[17] Moses, T. (ed.), eXtensible Access Control Markup Language (XACML) Version 2.0, Technical Report, OASIS Standard, 2005.

[18] D. J. Dougherty, C. Kirchner, H. Kirchner, A. S. de Oliveira, Modular access control via strategic rewriting, in: Proc. of ESORICS, pp. 578–593.

[19] D. Brewer, M. Nash, The chinese wall security policy, in: Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on, IEEE, pp. 206–214.

[20] N. Gruschka, N. Luttenberger, Protecting web services from DOS attacks by SOAP message validation, Security and Privacy in Dynamic Environments (2006).

[21] R. Bunge, S. Chung, B. E.-P. D., McLane, An operational framework for service oriented architecture network security, in: Proc. HICCS, p. 312.

[22] S. I. I. R. Room, XML Firewall Architecture and Best Practices for Configuration and Auditing, http://www.sans.org/reading_room/whitepapers/firewalls/xml-firewall-architecture-practices-configuration-auditing_1766, 2007.

[23] A. Singhal, T. Winograd, K. Scarfone, Guide to Secure Web Services, NIST Publication, 2007.

[24] netscaler, CITRIX NetScaler, http://www.citrix.com/english/ps2/products/product.asp?contentid=21679, 2010.

[25] ciscogw, CISCO ACE XML Gateway, http://www.cisco.com/en/US/products/ps7314/index.html, 2010.

[26] A. Knöpfel, B. Gröne, P. Tabeling, Fundamental modeling concepts, Wiley, West Sussex UK, 2005.

[27] D. von Oheimb, S. Mödersheim, ASLan++ - a formal security specification language for distributed systems, in: Prof. FMCO, pp. 1–22.