# On the Benefits of Decomposing Policy Engines into Components

Konstantin Beznosov

Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., Canada

beznosov@ece.ubc.ca

## ABSTRACT

In order for middleware systems to be adaptive, their properties and services need to support a wide variety of application-specific policies. However, application developers and administrators should not be expected to cope with complex policy languages and evaluation engines or to develop custom engines from scratch. In this paper, we discuss the benefits of policy engines designed as component frameworks with a mix of parameterized pre-built and custom logic composed to implement complex policies. To provide an example of such a design approach, we present an authorization architecture for ASP.NET Web services that has been implemented in a real-world system.

## Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design – *distributed systems*. D.2.11 [**Software Engineering**]: Software Architectures – *data abstraction, domain-specific architectures, information hiding, patterns*. D.2.13 [**Software Engineering**]: Reusable Software – *domain engineering, reuse models*.

## General Terms

Design, Experimentation, Security, Standardization.

## Keywords

Middleware, Policy, Authorization, Security, Architecture.

## 1. INTRODUCTION

Policy evaluation engines (or just "policy engines") provide decisions enforced by middleware mechanisms and services. The design and capabilities of policy engines also determine the corresponding administrative interfaces and languages.

The presupposition of this paper is that the research community needs to find better ways of designing policy engines that control the increasing flexibility of adaptive and reflective middleware architectures. Otherwise, the developers and owners of distributed applications will not be able to take the advantage of the flexible architectures if they are provided at the expense of administrative simplicity and usability.

To support flexible middleware behavior, policy decision logic is commonly constructed as generic, low-performance, and hard to administer complex policy engines in the attempt to support as many policy types as possible, or it is left to application developers to develop custom versions from scratch. After making a case that neither extreme is suitable, we argue that a better approach is to design policy engines as component frameworks that can be reconfigured and/or extended by re-arranging and/or adding/replacing individual components. Another key aspect of our approach is the pre-packaging of commonly used policy decision logic into policy components. We believe that such a hybrid approach gives the better of the two extremes. To show the feasibility of the proposed approach and to give an idea of how it could work, we describe an application of the approach to a specific policy type, authorization.

Being implemented in a real-world security solution, the described protection architecture makes ASP.NET Web service applications easier to integrate with organizational security infrastructure with a reduced effort on the side of Web service developers and owners. The architecture allows flexible configuration of the machine-wide authentication and authorization (A&A) functions, with the ability to override them for a sub-tree of the Web services. It supports a wide variety of authorization policies with the overhead proportional only to the complexity of the enforced policies. Furthermore, one can reuse authorization logic components by combining decisions from them according to predefined or custom rules. Although this architecture is specific to authorization, we hope that the general approach outlined in this paper is applicable to other types of policy decision logic.

The rest of this paper is organized as follows. Next section motivates the problem and discusses related work. Section 3 gives a general description of the proposed approach. In Section 4, we illustrate the approach on a concrete example of a policy engine. Examples of different configurations for the engine are provided in Section 5. We summarize and draw conclusions in Section 6.

## 2. PROBLEM MOTIVATION AND RELATED WORK

Some changeable aspects of middleware behavior are commonly implemented according to the principle of separation between mechanisms and policies, which was originally articulated by Levin et al. [11]. Policies are implemented by the mechanisms, where high-level


*3rd Workshop on Adaptive and Reflective Middleware*, Toronto, Canada.

mechanisms rely upon the low-level ones, creating a hierarchy under relation "is implemented using." In the cases when a simple and efficient interface between policies and mechanisms can exist, policy engines evaluate policies and provide decisions executed by the corresponding mechanisms.

A common example is access control, implementations of which follow the conceptual model of the reference monitor. The monitor can be viewed as a sum of two distinct but related parts: *enforcement function*, the mechanism, mediates access to resources by enforcing access control decisions obtained from the second part—*decision function*, the policy. Unless provisional authorizations [8] are employed, the decisions are binary, "deny" or "permit," facilitating the separation between the mechanism and the policy. Even though both functions are part of the *trusted computing base* [6] and therefore have to be carefully designed, implemented, and assured to perform according to the specification, the hardest one is the decision function due to the wide variety of policies and their flavors used in real-world applications [16-18].

Access control is one of the examples where implementations follow the decision-enforcement paradigm. Others are cryptographic data protection, scheduling, resource allocation, security audit, and object invocation dispatching.

The choices given to the developers and/or administrators of the distributed applications, with respect to the policy engines, are limited to two extremes. They are either pre-built policy engines that support particular, usually fixed, languages based on text or GUI, or programming interfaces for "plugging in" custom policy logic, which have to be constructed from scratch. As we argue in this section, both choices have significant limitations and drawbacks hampering their usefulness for real-world applications.

Among pre-defined policy languages and engines, some have limited scope and purpose, which enables efficient implementations, but limits the range of application domains where such implementations can be employed. Examples are COM+ and EJB authorization architectures described and compared in [7]. Both support only permission grouping into roles, whose scope is limited to an instance of a deployed application, and which cannot be organized in hierarchies or regulated by constraints, limiting both architectures to the basic Role-based Access Control (RBAC) reference model, $RBAC_0$, [17]. The applicability scope of such restricted policy logic is limited to a small class of application domains. As we explain in [7], the COM+ and EJB authorization policy engines are inadequate for enterprise-scale applications because of the poor scalability and the limited expressiveness.

Other pre-defined policy engines are designed according to "one size fits all" philosophy. Made very generic to support as many application domains as possible, their designs tend to sacrifice implementation efficiency to generality. Even worse, they are error prone to administer due to the complexity of the corresponding information models. An example is access control architecture of CORBA Security [14]. It was shown to support all major types of authorization policies: lattice-based mandatory and owner-based discretionary models [10] as well as role-based access control [4]. The versatile and generic CORBA access control architecture is so complex that the result of the author's attempt to capture it in a UML diagram for a research conference paper was criticized by a reviewer as a "monstrosity." Then, how could IT security administrators be expected to translate back and forth between application-specific policies and complex CORBA access control model?

The only other choice left to application developers and administrators is to implement custom policy evaluation engines using "plug in" programming interfaces available in some middleware architectures (e.g., CORBA Security replaceable interfaces, Java Authorization Contract for Containers [19] in J2EE) and policy engines (e.g., SiteMinder Authorization API [13]). This task could be notoriously hard and error prone for non-trivial.

Neither approach is quite what is needed. Pre-built policy engines are too limited, or too complex to administer and lack low-cost efficient implementations. "Do-it-yourself" extensions require prohibitively large development effort with high rate of costly errors. What could help? How can developers and/or administrators implement support for their custom policies without getting trapped in the complexities of the underlying generic languages and interfaces, or stepping on the slippery slope of "do-it-yourself" custom re-implementations of the policy logic?

This paper advocates an approach that can be viewed as an alternative to pre-built and custom policy evaluation engine approaches—policy engines designed as component frameworks (CF) that define a set of abstract interactions by which policy components cooperate in arriving to a policy decision. Although the benefits of component-based approaches are well known, little work has been published on component-based policy engines, including authorization ones.

Resource Access Decision (RAD) [5, 15] is one of few works that proposes an architecture for a component framework for an authorization service. RAD framework is composed of the following components: The *AccessDecisionObject* serves as the interface to RAD clients and coordinates the interactions between other RAD components. Zero or more *PolicyEvaluators* (PEs) perform evaluation decisions based on their access control policies that govern the access to a protected resource. The *DecisionCombinator* (DC) combines the results of the evaluations made by potentially multiple PEs into a final authorization decision by applying certain combination policies. The *PolicyEvaluatorLocator* (PEL), for a given access request to a protected resource, keeps track of and provides references to a DC and the PEs, which are collectively responsible for making the authorization decision to the request. The *DynamicAttributeService* (DAS) collects and provides dynamic attributes about the client in the context of the intended access operation on the given controlled resource. The protection architecture described in Section 4 follows RAD architecture more in spirit than in detail—rather as an architectural style. Our architecture also makes next step by decomposing the logic of retrieving authentication data and creating permissions into components (Section 4.3).

IBM Tivoli Access Manager [9] divides its authorization service into components similar to RAD. It supports custom versions of PEs and DAS but limits DC to the one provided with the implementation, and has no equivalent to PEL. In addition, the Access Manager architecture enables customization through the replacement of authentication data transformation logic, administration service, and the logic of retrieving data (e.g., "expense limit" information of customers) necessary for those application-specific policies that can only

be evaluated by the application. Because the logic of combining decisions from several PEs is restricted to simple voting with each PE having pre-configured vote weight, the capability of Access Manager architecture to support diverse policies via components composition is fairly limited. For instance, the policy from the second example (Section 5.2) could not be implemented through the composition of PEs and the use of the DC in the Access Manager.

# 3. WHAT COULD BE DONE ABOUT IT?

The premise of the approach proposed in this paper is the following. Although application domains and even individual applications differ in the specifics of their policies, common elements are often present in many policy instances. For example, many access control decisions are based on common factors such as the roles of the users, their group memberships, and the locations of the computers from which access is attempted, to name a few. The difference is often found in the weight of these factors in the final decisions as well as in those factors that are specific to individual applications, e.g., relationship between a doctor and a patient [1]. Therefore we propose the use of component frameworks for constructing policy engines.

Continuing the example with access control, the logic of policy-driven decisions based on roles, groups, client IP addresses, etc., could be pre-built into authorization and other related modules and packaged as components. Application developers/administrators can then compose these modules into specific policy engines without having to implement the engines from scratch. On the other hand, with the sufficient flexibility of the policy engine architecture, modules that implement support for unusual factors should be easy to implement and "plug in."

In CF-based policy engines, the complexity and the implementation efficiency are largely determined by the complexity of the required policies and not by the (used and unused) capabilities of the policy engine. In addition, tailoring the evaluation logic to the idiosyncrasies of the specific application (domain) would not necessarily require complete re-implementation; incremental addition of a custom-made component could suffice.

# 4. FEASIBILITY OF THE APPROACH

To support our position that designing policy evaluation logic in the form of component frameworks, in which individual policy modules are easy to re-use, replace, and augment by combining with other such modules, is feasible and advantageous, we describe one instance of this approach.

The protection architecture presented in this section makes ASP.NET Web service applications easier to integrate with organizational security infrastructure with a reduced effort on the side of Web service developers. The architecture is flexible because it allows configuring machine-wide authentication and authorization functions, and overriding them for a sub-tree of the Web services (up to an individual Web service) in the directory-based ASP.NET hierarchy. Its extensibility is revealed through the support of wide variety of authentication and authorization (A&A) logic. Furthermore, one can reuse other instances of such logic by combining authorization decisions from them according to predefined or custom rules.

The following sub-sections describe the most relevant elements of the architecture, whereas the detailed description can be found in [3].

## 4.1 General Structure

To integrate with ASP.NET run-time, the architecture takes advantage of the ASP.NET generic interception mechanism, SOAPExtension [12], intended for custom processing of SOAP messages. Our interceptor performs initial extraction, formatting, and other preparation of HTTP and, contained in them, SOAP messages, passing the data to the decision A&A logic, and enforcing authorization decisions. The object of discussion in this paper is the authorization policy evaluation engine.

## 4.2 Authorization

The structure of the authorization-related elements follows RAD and Attribute Function [2] architectural styles. An authorization decision is reached in a three-step process.

Initial decisions are made by zero or more *PolicyEvaluator*s. The simplest PE is one that always returns same decision, e.g., "deny," "permit," depending on its static configuration. More interesting PEs grant access based on IP address of the request sender, name of the Web service target and its methods, and decisions provided by an enterprise authorization server. The strength of RAD architectural style is in the support of fairly sophisticated[1] authorization policies without the need for complex authorization engines. The support is achieved by combining run-time decisions from several simple PEs into one at the second step, performed by a *DecisionCombinator*. Similarly to PEs, common variations of combination logic are provided in pre-built DCs with the ability for developers to "plug" custom implementations.

The authorization process continues to its third stage in order to achieve *fail-safe defaults*, in the cases when a DC experiences a failure, and due to a design or implementation error, does not come to a binary decision, During this stage, the interceptor, which originally delegated the process to the corresponding DC, renders any decision, except "permit," received from the DC to "deny" and thus reaching authorization verdict. If access has been denied, the corresponding exception with the configurable explanation message is thrown to the ASP.NET run time, which translates it into an appropriate SOAP exception message.

Besides credentials, obtained from the SOAP message, the corresponding HTTP request, or the underlying communication channel, PEs are supplied with other information related to the request. All this information is constructed into a *permission*. The authorization process results in a decision whether a permission should be granted to a (potentially compound) subject given subject's credentials. If so, then the interceptor passes control to ASP.NET which activates the corresponding Web service implementation and passes to it the request contained in the SOAP message. It is the construction of the permission that furthers the flexibility and extensibility of the architecture.

## 4.3 Permission Construction

To support the flexibility and extensibility of the architecture, we designed permission construction out of four distinct

[1] See [1] for an example.

elements, shown at the bottom of Figure 1: target name, domain name, target attributes and method name. The construction of permissions is done by a default permission factory, which can be replaced by a custom implementation possibly producing permissions of other format and content.

## 4.4 Replaceable Parts

The flexibility and extensibility of the architecture are achieved via designing most of its elements as components. Any of the black boxes in Figure 1 can be replaced by a version that comes with the implementation or by a version produced by Web service developers or owners.

Custom versions of the grey boxes are subject to the control by those modules that create them. Other architectures, e.g., CORBA Security, also make some of their parts replaceable. The novelty of our approach is in the level of replaceable parts' granularity. In CORBA Security, for instance, authorization logic (encapsulated in *AccessDecision* interface) has to be replaced as a whole, whereas in our architecture, one can selectively replace specific PEs and/or a DC.

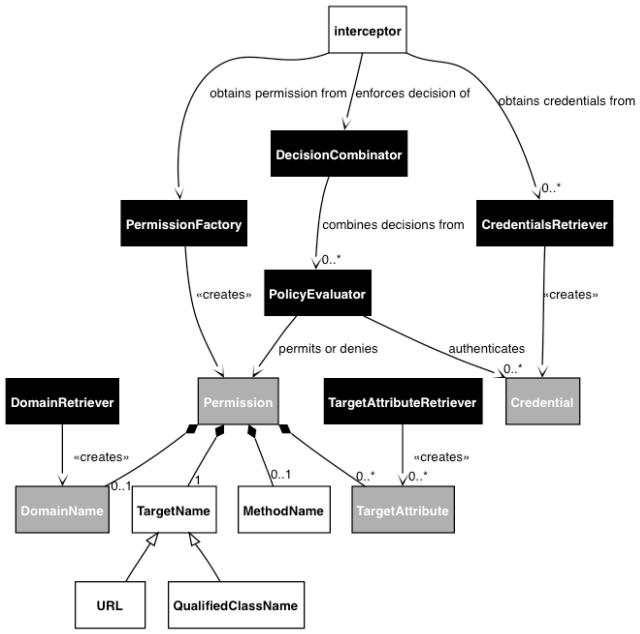


**Figure 1**: **Key elements of the authorization policy engine architecture. (replaceable in black, modifiable in grey).**

# 5. Examples

To demonstrate the ability of our architecture to be customized through different compositions of its components, we provide examples of implementing two different policies.

## 5.1 Example 1: University Course Web Service

Consider a simplified hypothetical application that enables online access to university courses as web services. Let us assume that the following is a relevant to the example fragment of the application security policy to be enforced:

**Policy 1**:

1. All users should authenticate using HTTP-BA.
2. **Anybody** can lookup course descriptions.
3. **Registration clerks** can *list students* registered for the course and (*un)register* students.
4. The **course instructor** can *list registered students* as well as *manage course assignments* and *course material*.
5. **Registered** for the course **students** can *download assignments* and *course material*, as well as *submit assignments*.

Given that each course is represented by a separate instance of a web service, the following is a configuration of our architecture that enables the enforcement of Policy 1. It is illustrated in Figure 2 with custom-built modules in black.

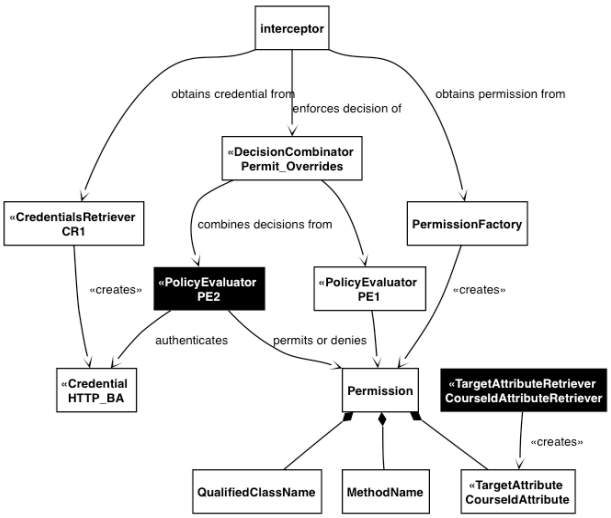


**Figure 2: Configuration 1.**

**Configuration 1**:

- An HTTP-BA *CredentialRetriever* $CR_1$ extracts the user name and password from the corresponding HTTP request.
- A custom *TargetAttributeRetriever* that provides the course number in a form of an attribute, e.g. CourseId=EECE412.
- The default *PermissionFactory* is configured to compose permissions with the qualified .NET class name, as a *TargetName*, the method name, and the attributes provided by the custom retriever. Here is an example: 'ca.ubc.CourseManagment.SimpleCourse/CourseId=EECE 412/GetDescription'. No domain name is used in this configuration.
- A pre-built *PolicyEvaluator* $PE_1$ that grants permissions to any request on public methods. In the case of Policy 1, there is one public method, GetCourseDescription.
- A custom *PolicyEvaluator* $PE_2$ programmed and configured to make authorization decisions according to the rules informally described as follows:
  1. Permit registration clerks to access methods 'ListStudents', '(Un)RegisterStudent'.
  2. Permit users in role 'instructor' whose attribute 'CourseTaught' contains the course listed in Permission.TargetAttributes.CourseId to list students, manage course assignments and material.
  3. Permit users in role 'student' whose attribute 'RegisteredCourses' contains the course listed in

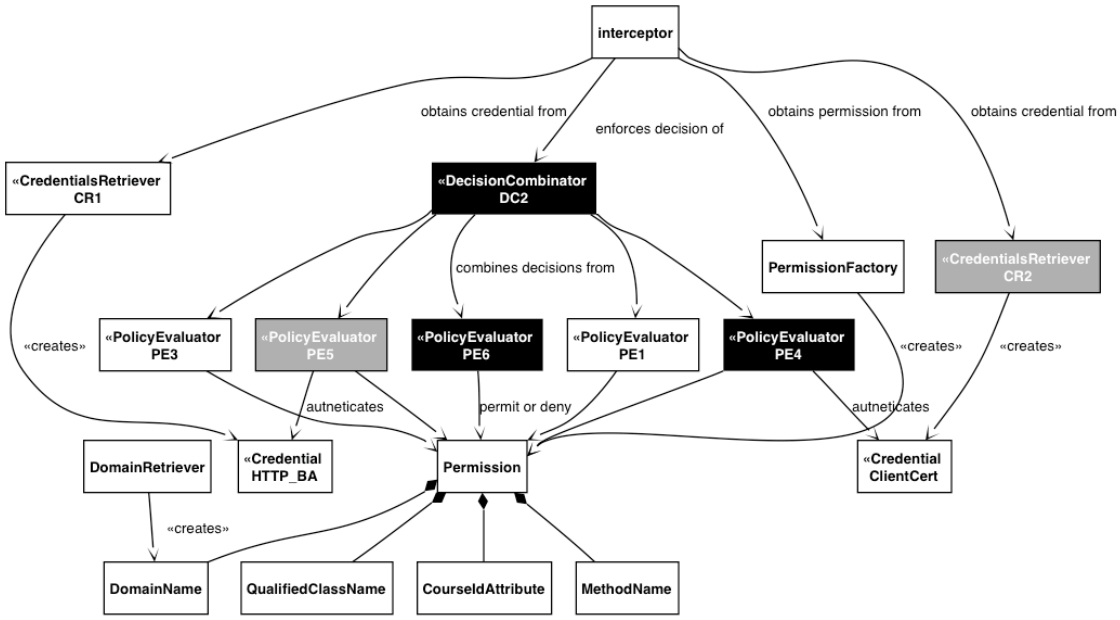


**Figure 3**: **Configuration 2. Custom-built components have black fill color. Generic components from third-party vendors have gray fill color.**

Permission.TargetAttributes.CourseId to list students, manage course assignments and material.

- A pre-built *DecisionCombinator* of type *Permit Overrides*, which grants access if either PE grants access.

## 5.2 Example 2: Human Resource Web Service for an International Organization

Now consider a multinational company. Each division has its own department of human resources (HR). The company rolls out a web service application to provide online access to employee information. Each division has Web services providing HR information of that division. The following policy for accessing this application is to be enforced.

**Policy 2**:

1. Only users within the *company's intranet* or those who access the service over SSL and have valid X.509 certificates issued by the company should be able to access the application.
2. **Anybody** in the company can *look up* any employee and *get essential information* (e.g., contact information, title, and names of the manager and supervised employees).
3. **HR Employees** can *modify contact information* and *review salary* of any employee from the same division.
4. **Managers of HR** departments can *modify any information* about the employees of the same department.

**Configuration 2**:

- Same *CredentialsRetriever* $CR_1$ as in Example1.
- Another *CredentialRetriever* $CR_2$ obtains an SSL client certificate from the corresponding HTTPS connection.
- A pre-built simple *DomainRetriever* that always returns same statically configured domain name. The domain name designates the division for which HR information is served by the web service instance, e.g. 'Japan'.
- The default *PermissionFactory* is configured to compose permissions with the domain name, .NET class name, as a target name, and the corresponding method name. No target attributes are used in this case. Here is an example: 'Japan/com.mega-foo.EmployeeInfo/GetContactInfo'.
- Same *PolicyEvaluator* $PE_1$ as in Example 1. In this case, the public methods are: FindEmployee, GetEmployeeInfo, GetEmployeeManager, GetSupervisedEmployees.
- A pre-built *PolicyEvaluator* $PE_3$ that permits access to any request made from a machine with an IP address in the range of the company's intranet addresses.
- A custom-built *PolicyEvaluator* $PE_4$ that permits access to any request made by a user with valid X.509 certificate issued by the company. The certificate is retrieved by $CR_2$.
- A generic RBAC *PolicyEvaluator* $PE_5$ that permits invocation of different methods based on the user role:
  1. Any user with role 'hr employee' can invoke methods that modify contact information and review salary.
  2. Any user with role 'hr manager' can invoke methods permitted to users with role 'hr employee' as well as methods that modify employee's salary, title, and names of the manager and supervised employees.
- A custom built *PolicyEvaluator* $PE_6$ that permits access to any authenticated user, whose attribute 'Division' has the same value as the domain in the permission.
- A custom-built *DecisionCombinator* $DC_2$ which grants access according to the following formula: $(PE_3 \vee PE_4) \wedge (PE_1 \vee (PE_5 \wedge PE_6))$. That is, a request is permitted only to intranet users or those with valid compnay's certificate $(PE_3 \vee PE_4)$, provided that either the requested method is public $(PE_1)$ or an authorized HR person is accessing a record of the employee from same division $(PE_5 \wedge PE_6)$.

The high degree of the architecture composability allows re-using two pre-built ($PE_1$ & $PE_3$). Even though configuration 2 has 3 more PEs and one more CredentialRetriever than configuration 1, as shown in Figure 3, there are only three components ($DC_2$, $PE_4$, and $PE_6$) that have to be custom-built. Among them, $PE_4$ is simple to build using certificate validation tools and libraries, and $PE_6$ requires marginal effort. $DC_2$ can be implemented in one 'if' structure. Two other ($PE_5$ and $CR_2$) are generic and can be supplied by third-party vendors.

# 6. SUMMARY AND CONCLUSIONS

In this paper, we consider the problem of constructing flexible policy engines for adaptive middleware systems. Governing the behavior of various middleware mechanisms and properties, the engines have to support a wide variety of policy types and their variants. This flexibility is commonly achieved either through the construction of generic, low-performance, and hard to administer complex policy engines in the attempts to support as many policy types as possible, or leaving application developers to implement custom engines from scratch. After making a case that neither extreme is suitable, we propose a hybrid approach.

We argue that a better approach is to design policy engines as component frameworks that can be reconfigured and/or extended by re-arranging and/or adding/replacing individual components. Another key aspect of our approach is the pre-packaging of commonly used policy decision logic into policy components. We believe that such an approach is better than the two extremes. It avoids the complexity and unnecessary run-time and administrative overhead of generic policy engines. At the same time, application developers have a simpler job of creating custom policies by rearranging existing policy components and/or incrementally adding new ones. To show the feasibility of the proposed approach and to give an idea of how it could work, we describe an application of the approach to a specific policy type, authorization.

The protection architecture described in this paper makes ASP.NET Web service applications easier to integrate with organizational security infrastructure. The architecture allows configuring machine-wide A&A logic, and overriding them for a sub-tree of the Web services. Following the RAD [5] and AF [2] architectural styles, it supports a wide variety of authorization policies. The architecture has been implemented in a real-world security solution.

There is an alternative to complex, generic, and therefore expensive to build, execute, and administer, general-purpose authorization engines. This alternative is lightweight, simple to construct, and inexpensive to run authorization modules, each of which is dedicated to evaluating very specific type of authorization rules. Other lightweight specialized modules combine decisions from these modules. As a result, for every distinct authorization policy a specialized version of the authorization engine is composed out of such modules. Even more, multiple specialized engines can govern access to the Web services co-located in one ASP.NET container. Although this architecture is specific to authorization, we hope that the general approach outlined in this paper is applicable to the other types of policy decision logic.

# 7. REFERENCES


[1] Barkley, J., Beznosov, K. and Uppal, J., Supporting Relationships in Access Control Using Role Based Access Control. in *Fourth ACM Role-based Access Control Workshop*, (Fairfax, Virginia, USA, 1999), 55-65.

[2] Beznosov, K., Object Security Attributes: Enabling Application-specific Access Control in Middleware. in *4th International Symposium on Distributed Objects & Applications (DOA)*, (Irvine, California, USA, 2002), Springer-Verlag, 693-710.

[3] Beznosov, K., Protecting ASP.NET Web Services: Experience Report. in *preparation*, (2004).

[4] Beznosov, K. and Deng, Y., A Framework for Implementing Role-based Access Control Using CORBA Security Service. in *Fourth ACM Workshop on Role-Based Access Control*, (Fairfax, Virginia, USA, 1999), 19-30.

[5] Beznosov, K., Deng, Y., Blakley, B., Burt, C. and Barkley, J., A Resource Access Decision Service for CORBA-based Distributed Systems. in *Annual Computer Security Applications Conference*, (Phoenix, Arizona, USA, 1999), IEEE Computer Society, 310-319.

[6] DOD, DoD 5200.28-STD: Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC), Department of Defense, 1985.

[7] Hartman, B., Flinn, D.J., Beznosov, K. and Kawamoto, S. *Mastering Web Services Security*. John Wiley & Sons, Inc., New York, 2003.

[8] Jajodia, S., Kudo, M. and Subrahmanian, V.S., Provisional authorizations. in *1st Workshop on Security and Privacy in E-Commerce*, (Athens, Greece, 2000), Kluwer Academic Press, Boston, MA.

[9] Karjoth, G. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and Systems Security*, *6* (2). 232-257.

[10] Karjoth, G. Authorization in CORBA Security. *Journal of Computer Security*, *8* (2/3). 89-108.

[11] Levin, R., Cohen, E., Corwin, W., Pollack, F. and Wulf, W., Policy/mechanism separation in Hydra. in *ACM Symposium on Operating Systems Principles*, (Austin, Texas, United States, 1975), ACM Press, 132--140.

[12] Microsoft. Altering the SOAP Message Using SOAP Extensions, 2002.

[13] Netegrity. SiteMinder Developer's API Guide, Netegrity, Waltham, MA, 2000, 410.

[14] OMG. CORBAservices: Common Object Services Specification, Security Service Specification v1.8, Object Management Group, document formal/2002-03-11, 2002.

[15] OMG. Resource Access Decision Facility, Object Management Group, 2000.

[16] Sandhu, R., Access Control: The Neglected Frontier (Invited Paper). in *First Australasian Conference on Information Security and Privacy*, (Wolongong, Australia, 1996).

[17] Sandhu, R., Coyne, E., Feinstein, H. and Youman, C. Role-Based Access Control Models. *IEEE Computer*, *29* (2). 38-47.

[18] Sandhu, R.S. Lattice-Based Access Control Models *IEEE Computer*, 1993, 9-19.

[19] Sun. Java Authorization Contract for Containers, 2002.