Here's Your Lego™ Security Kit: How to Give Developers All Protection Mechanisms They Will Ever Need

Konstantin Beznosov

Department of Electrical and Computer Engineering, University of British Columbia
beznosov@ece.ubc.ca

Abstract. By presenting a protection architecture for ASP.NET Web services, this paper demonstrates the feasibility of creating middleware mechanisms in the form of composable, flexible, and extensible building blocks. Like Lego™ constructor parts, such blocks enable the reduction of the effort of constructing, extending, and adjusting the application properties and middleware services in response to requirements or environment changes.

1 Introduction

The main premise of this paper is that the developers and owners of distributed applications need and can be provided with three things: 1) Lego™-like reusable and versatile building blocks, 2) middleware architectures and tools for composing useful customized solutions out of such blocks, and 3) the means of creating their own inexpensive and error-proof building blocks. They could then create custom distributed applications suitable to their needs and environments, while avoiding costly reinvention and reconstruction of generic and, more often than not, quite complex functionality common across applications. And we are not referring to the business logic, which could arguably be included in the list. The focus is on the nonfunctional properties and services (fault tolerance, performance, security, etc.) of distributed applications.

The above needs have been determined from the author's experience of working for end-user, consulting, and vendor organizations. Working on the end-user side showed that no vendor could ever satisfy all requirements for customizing their solutions to our needs and constraints. Vendors' customization mechanisms required too much effort and expertise from in-house developers. Experience as a consultant, product developer and architect gave convincing evidence that this problem was common to many end-user organizations.

To demonstrate that useful building blocks, architectures, and extension means can indeed be provided for customizing nonfunctional properties of distributed applications without demanding seasoned expertise in the subject matter from application developers, we present an *authentication and authorization (A &A)* architecture for ASP.NET Web services. This architecture, we believe, features all three desired characteristics. It builds on the results of several years of applied research and practical

T. Gschwind and C. Mascolo (Eds.): SEM 2004, LNCS 3437, pp. 3 – 18, 2005.

experience, giving the hope that similar architectures can be developed for easy customization of other properties and services for distributed applications.

The paper is organized as follows: section 2 provides background and discusses related work; Section 3 explains technical motivations for the architecture and gives its overview; Section 4 highlights those design decisions that made the architecture easy to customize; discussion is in Section 5; and we conclude with Section 6.

2 Background and Related Work

Research on composition and customization for middleware has been largely focused on three areas: core functionality; domain-specific properties and characteristics; and middleware services. Research in core functionality concentrates on data (un)marshaling, invocation dispatching, object life-cycle, data transport, etc. (TAO [1], Quarterware [2], COMERA [3], Spring [4]). Examples from the work in domain-specific properties and characteristics are real-time [5], load-balancing [6], QoS [7, 8], performance and consistency [9]. Our work is on composable and customizable A&A mechanisms and belongs to middleware services research, which concentrates on such services as event notification [10], transactions and concurrency [11, 12], and security.

Work on customizable security mechanisms in middleware has been conducted at least since DCE [13]. A wider known example is CORBA, which has a Security Service [14] architecture that enables customization by supporting interceptors as well as making authorization and audit decision objects, security context and some other elements replaceable. However, because the granularity of CORBA Security replaceable parts is too coarse it takes too much effort to customize the service. This drawback can also be viewed as low degree of *composability*. Besides DCE and CORBA, other examples of architectures with replaceable security logic but low degree of composability are more modern JAAS [15], Java Authorization Contract for Containers architecture [16], and Legion [17]. Our approach achieves fine granularity of the replaceable parts and therefore a higher degree of composability.

What our approach (intentionally) leaves unanswered is how to express A&A policies and map them into a composition of A&A building blocks. Andersen et al. [18] approach the problem from the other end and propose "programmable security" approach that uses Obol language to "program" middleware security protocols without addressing the issue of translating such programs into compositions of specific elements of the middleware security architecture.

Design of the authorization mechanism described in this paper is largely based on the Resource Access Decision (RAD) architecture [19, 20], which we follow more in the spirit than in detail—rather as an architectural style. Briefly reviewed in Appendix A, RAD is one of the first attempts to compose and customize authorization logic out of simpler parts.

Although, neither RAD nor this work address the issue of conflicts that could arise as a result of authorization logic composition, several solutions have been proposed elsewhere. Jajodia et al. [21] have proposed an access control model in which inconsistencies among authorizations can be resolved using rules. The framework for access control policy enforcement developed by Siewe et al. [22] allows multiple poli-

cies to be enforced through policies composition. It provides a way to specify complex policies and to reason about their properties.

3 Architecture Motivation and Overview

The ASP.NET container is a popular hosting environment for Web services built and run on Microsoft Windows and .NET platforms. However, the ASP.NET security architecture [23], as provided out-of-the-box is not sufficiently flexible and extensible to be adequate for enterprise applications. As we describe in [24], ASP.NET supports limited authentication and group/user-based authorization, both bound to Microsoft proprietary technologies. If an application needs to be protected with enterprise A&A services, the developers have two options: The first, is to develop home-grown container security extensions, which are hard for average application developers to get right. The second option is to program the security logic into the Web service business logic, but the resulting application is costly to evolve and support. In both cases, the development of security-specific parts by average application developers is commonly believed to result in high vulnerability rates due to security-related bugs that are hard to avoid and catch.

Due to its flexibility and extensibility, the protection architecture described in this paper makes ASP.NET 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 of machine-wide authentication and authorization functions, and overriding them for a subtree of the Web services (up to an individual Web service application) in the directory-based ASP.NET hierarchy. Its extensibility is revealed through the support of wide variety of A&A logic, as long as the logic can be programmed as a .NET class and/or accessed (possibly via a proxy) through a predefined .NET API. Furthermore, one can reuse other instances of such logic by combining authorization decisions from them according to predefined or custom rules.

4 The Architecture

The architecture details are described elsewhere [25]. This section focuses mainly on those features of the architecture that enable the composition of more complex A&A functionality from basic, reusable, building blocks. There are five features:

1. the separation of A&A enforcement logic from the decision logic,
2. the employment of the RAD architecture style, which makes creation of custom authorization decision logic easier and avoids the need for a general-purpose policy evaluation engine,
3. flexible configuration-driven construction of the authorization decision information,
4. fine-grained replaceable modules that enable support for a wide range of A&A functionalities, and
5. the support for the scalability, extensibility, and reusability in the configuration part of the architecture.

While most of these features have been already reported individually in the literature, the novelty of our approach is in achieving new characteristics of middleware protection mechanisms by exploiting and combining these features.

4.1 Separation of Enforcement and Decision Logic

To integrate with ASP.NET run-time, the architecture takes advantage of the ASP.NET interception mechanism, *SOAP Extension* [26], intended for additional processing of SOAP messages. Although this mechanism is specific to ASP.NET, other modern middleware technologies (e.g., CORBA, EJB, RMI) can intercept requests or even individual messages [27-30]. Hence, the reliance on the existence of an interception point in the request invocation chain does not limit our approach or makes it specific to ASP.NET.

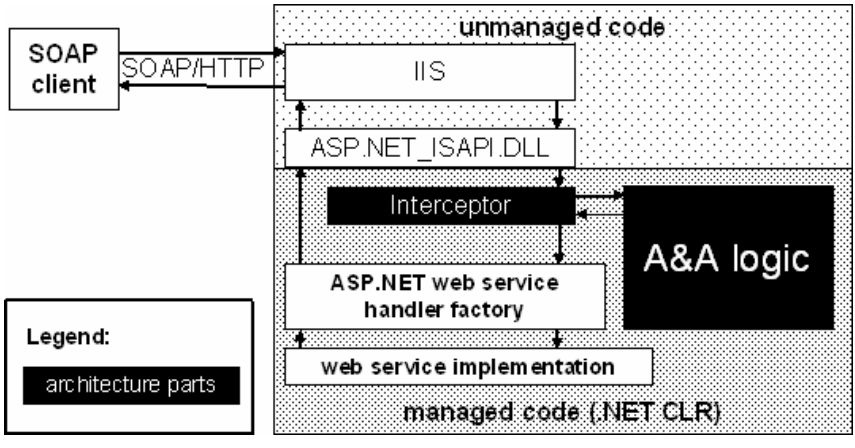

Fig. 1. General organization of the architecture into an interceptor

As shown in Figure 1, our custom version of SOAP Extension module (labeled "interceptor") performs initial extraction, formatting, and other preparation of HTTP requests and contained in them SOAP messages, passing the data to the decision A&A logic, and enforcing authorization decisions. Through the separation of the enforcement and decision functions, we were able to make the enforcement policy neutral and common to all Web services, while allowing the latter to be customizable to each application. The customizable functionalities are authentication and authorization.

Authentication is commonly divided into two phases: retrieving authentication data and validating it. *CredentialsRetriever* objects specialize in retrieving authentication data, whereas validation follows lazy strategy and is left to the authorization phase. Each retriever implementation is responsible for extracting particular data types from appropriate locations. Authentication data and retrievers are represented in a uniform fashion as implementations of *Credential* and *CredentialsRetreiver* interfaces accordingly. This extensibility enables support for diverse authentication policies. For instance, in the same ASP.NET container, one application might use only HTTP basic

authentication with username and password (HTTP-BA) over SSL, whereas another could require client SSL certificate and a security token in the SOAP message header to be present for successful authentication.

4.2 Employment of the RAD Architectural Style

The structure of the authorization-related elements of the architecture follows RAD style, which enables the composition of more complex authorization policies out of simpler ones. A brief overview of RAD is provided in Appendix A.

An authorization decision is reached in a three-step process made by evaluators, combinator, and interceptor. Initial decisions are made by zero or more predefined or custom authorization modules referred as *Policy Evaluators* (PEs). The strength of RAD architectural style is in the support of fairly sophisticated authorization policies (see [31] for an example) 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 *Decision Combinator* (DC).

Similarly to PEs, common variations of combination logic are provided in pre-built DCs with the ability for developers to "plug in" custom implementations. To appreciate the power of DC&PEs approach, consider a composition of "All Permits Required" DC with a role-based access control (RBAC) [32] PE. If an application owner decides to further restrict access to a particular range of IP addresses, he or she can do so by adding a PE that authorizes IP addresses, instead of modifying fairly complex logic of the RBAC PE. The result is shown in Figure 2. Support for policies in which PEs might have different priorities, is enabled through the use of PE names so that a DC can discriminate between them.

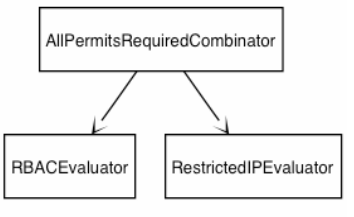

Fig. 2. Resulted configuration after adding the PE, which restricts access based on the sender's IP address

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 reaches an authorization verdict. If access has been denied, the corresponding exception is thrown to the ASP.NET run-time, which translates it into an appropriate SOAP exception message.

4.3 Adaptable Information for Authorization Decisions

Besides credentials, PEs are supplied with other request-related information, which is constructed into a *permission*. Thus, the authorization process determines whether a permission should be granted to a subject given its credentials. It is the adaptable construction of the permission that furthers the composability and customizability of the architecture. A permission is constructed out of four distinct elements, as shown in Figure 3. Examples are provided in Table 1 at the end of this subsection.

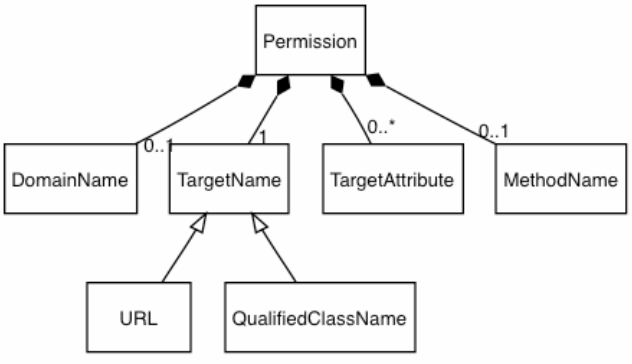

Fig. 3. Elements of the permission generated by the default permission factory

1. *TargetName:* the name of the target Web service can be represented by either the URL or the .NET class name of the service implementation. By using the .NET class name instead of the URL, all instances of a Web service application can share the same authorization policy rules.
2. *DomainName:* the use of the domain classifier is borrowed from CORBA Security [14] architecture, whose policy domains support different security requirements for implementations of same interfaces. In our architecture, optional domains allow discrimination between those same implementations of a Web service that have different access control requirements. Another intended purpose of domains is to allow a logical grouping of several Web services, perhaps so that they can share an authorization server or its policy database.
3. *TargetAttributes:* further differentiation among Web service instances is achieved through an optional list of name-value pairs holding target attributes. For example, a Web service representing a university course could have the course Id as one of its attributes. The use of target attributes reduces the need for mixing authorization and other security logic with business logic. These application-specific attributes and the mechanism for obtaining them are directly based on our prior work on Attribute Function (AF) [33, 34], overview of which is provided in Appendix B.
4. *MethodName:* since ASP.NET supports only RPC semantics, acceptable SOAP messages have to specify the method of the corresponding .NET server class.

Table 1 shows examples of permissions. 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.

Table 1. Examples of permissions

Permission Example	Explanation
http://foobank.com/bar.asmx	Only the URL is used
com.foobank.ws.Sbar/m1	Class and method names
D1/com.foobank.ws.Sbar /m1	Same but in domain "D1"
com.foobank.ws.Sbar/owner=smith	Class name and attribute
D1/com.foobank.ws.Sbar/owner=smith/m1	Domain, class, attribute, method

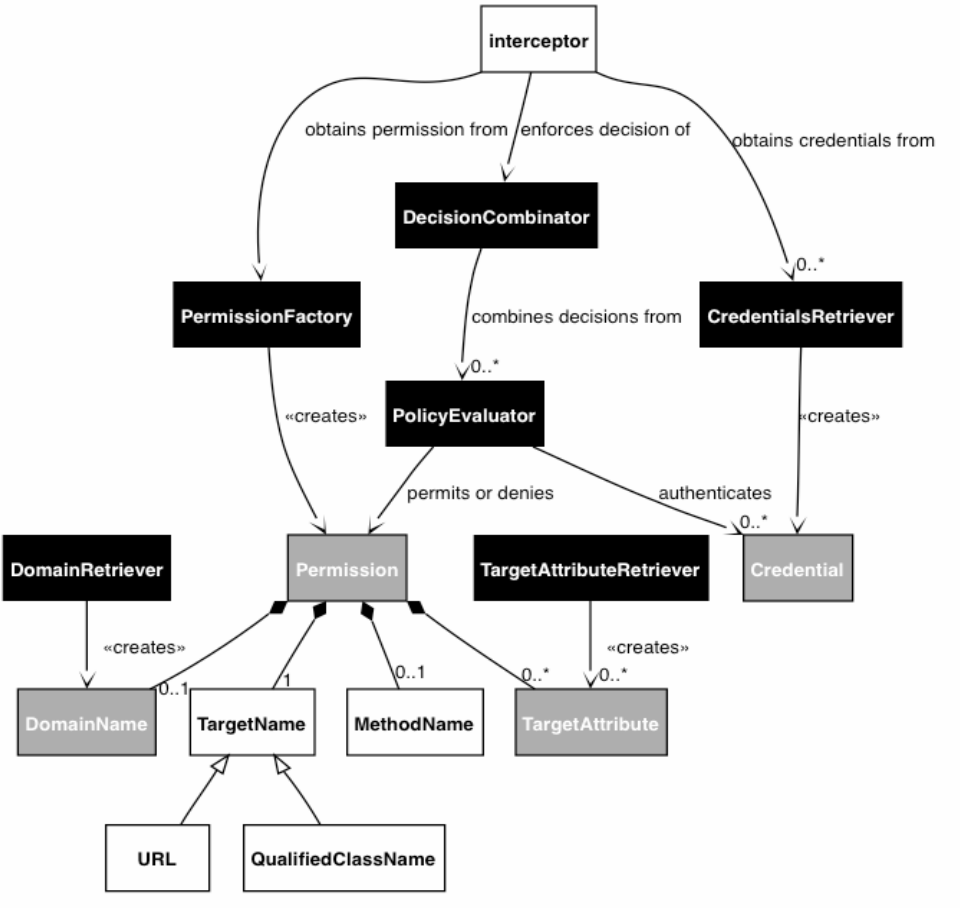

Fig. 4. Key elements of the architecture: black elements are replaceable, and grey elements are modifiable by their creators

4.4 Fine-Grained Replaceability

The flexibility and extensibility of the architecture is achieved in part by designing most of its elements to be replaceable. Any of the black boxes in Figure 4 can be replaced by a version that comes with the implementation or by a version produced by application 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 the level of replaceable parts' granularity. In CORBA Security, for instance, authorization logic has to be replaced as a whole, whereas in our architecture, one can selectively replace specific PEs and/or a DC. Additionally, each Web service in the same container can be protected by a different set of replaceable elements, which is not the case with CORBA Security implementations.

To demonstrate the ability of our architecture to be customized through different compositions of black-box implementations we provide examples of implementing two different policies.

4.4.1 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 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.

Configuration 1:

- An HTTP-BA *CredentialRetriever* CR_1 extracts the user name and password from the HTTP request that carried the corresponding SOAP request.
- A custom *TargetAttributeRetriever* 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 class name of the .NET class, as a *TargetName*, the corresponding method name, and the attributes provided by the custom retriever. For example: 'ca.ubc.CourseManagment.SimpleCourse/CourseId=EECE412/GetDescription'. No domain name is used in this configuration.

- A prebuilt *PolicyEvaluator* PE_1 grants permissions to any request on publicly accessible methods. In the case of Policy 1, there is one public method, GetCourseDescription.
- A custom *PolicyEvaluator* PE_2 is programmed and configured to make authorization decisions according to the rules informally described as follows:

1. Permit users in role 'registration clerk' to access methods 'ListStudents', 'RegisterStudent' and 'UnregisterStudent'.
2. Permit users in role 'instructor' whose attribute 'CourseTaught' contains the course listed in Permission.TargetAttributes.CourseId to list registered students, manage course assignments and material.
3. Permit users in role 'student' whose attribute 'RegisteredCourses' contains the course listed in Permission.TargetAttributes.CourseId to list registered students, manage course assignments and material.

Note that user roles and other attributes are retrieved by the PE during or after it validates the credential received from HTTP-BA *CredentialRetriever.* This step is not discussed since it is very specific to the particular student and employee databases used by the university and is irrelevant here.

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

4.4.2 Example 2: Human Resource Web Service for International Organization

Now consider a multinational company that has its divisions in Japan, Canada, Austria, and Russia. Each division has its own department of human resources (HR). The company rolls out a Web service application in all of its divisions to provide online access to employee information. Each division has one or more Web services providing HR information of that division. The company establishes the following security policy for accessing this application.

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* about her/him (e.g., contact information, title, and names of the manager and supervised employees).
3. **Employees of HR** departments can *modify contact information* and *review salary information* 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 Example 1.
- Another *CredentialRetriever* CR_2 obtains an SSL client certificate from the corresponding HTTPS connection.

- A prebuilt 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, qualified class name of the .NET class, as a target name, and the corresponding method name. No target attributes are used in this case. For example: 'Japan/com.mega-foo.EmployeeInfo/GetContactInfo'.
- Same prebuilt *PolicyEvaluator* PE_1 as in Example 1. In the case of Policy 2, there are four public methods: FindEmployee, GetEmployeeInformation, GetEmployeeManager, GetSupervisedEmployees.
- A prebuilt *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. This certificate, if available, is retrieved by CR_2.
- A generic RBAC *PolicyEvaluator* PE_5 that permits invocation of different methods based on the role of the user:
  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*, 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 company'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 reusing two prebuilt (1 & 3) and one generic (RBAC) PE (5) out of five. Among the other two, PE_4 is simple to build using certificate validation tools and libraries, and PE_6 requires marginal effort. The DC can be implemented in one 'if' structure.

4.5 Configuration Scalability, Extensibility, and Reuse

Extensible and scalable configuration turned out to be critical in order for our architecture to support the composition of more complex A&A functionality from basic, reusable, building blocks, and, at the same time, carry low administration or run-time overhead. We developed a simple hierarchical language for defining and configuring various elements of the A&A decision logic as well as the protection policies composed of them. The relationships among these elements and the policies are shown in Figure 5.

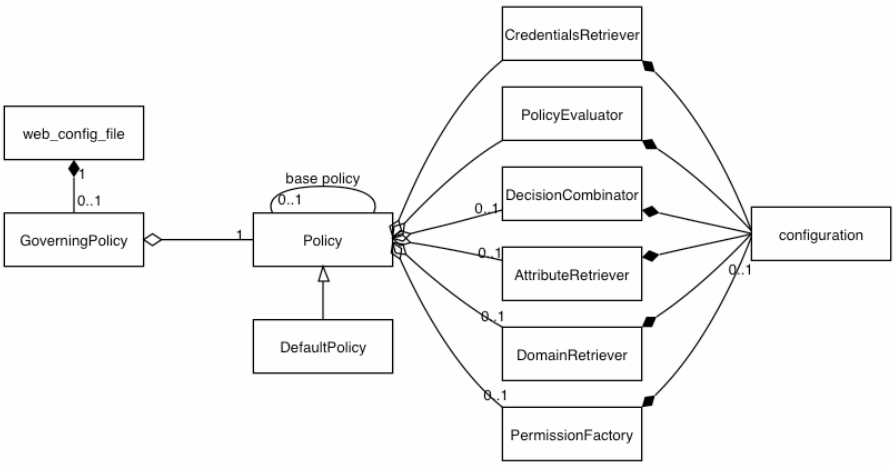

Fig. 5. Simplified model of the configuration elements with default cardinality "0..*"

A protection policy can simply be viewed as a collection of specific credential retrievers; Pes; DC; target and domain and target attribute retrievers; as well as a permission factory, which is defined in other sections of the configuration. Since all these elements are defined independently of the policies and have unique names, they can be referenced by more than one policy. *Governing Policy* (GP) specifies which particular policy is used for controlling access to a Web service. Thus, multiple policies can be prepackaged and used for quickly switching the behavior of the protection mechanisms from one predefined mode to another.

Illustrated in Figure 6, the hierarchal nature of web.config parsing semantics enables a high degree of scalability without losing a fine level of granularity in the control over subsets of (or individual) Web services. The GP defined in the web.config of the ASP.NET root determines the protection of all those Web services, for which no web.config file between the service and the root directory overrides it.

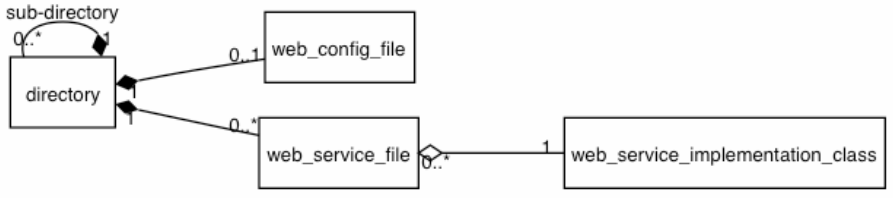

Fig. 6. The association among Web services, their implementations, directories and configuration files

The configuration extensibility and reusability is achieved through two design decisions: First, any web.config file down in the ASP.NET directory hierarchy can override GP, or define any new element, including new policies as long as the name of

this element has not been used in an ascendent web.config.[1] Second, to reduce the amount of effort required for creating policy variations, we also introduced single inheritance mechanism for policy definitions. This way, a policy could reuse most of another policy's definition and override just few elements.

5 Discussion

Besides the practical value for developers and owners of Web services hosted by ASP.NET containers, the work on the A&A architecture demonstrated two points worth of discussion: First, the architecture of protection mechanisms for distributed applications can be designed as a collection of easy-to-create building blocks with multiple places for extending and altering the overall behavior. Hopefully, this work will encourage middleware and software engineering communities to look into the feasibility of similar designs for other mechanisms and services.

Second, there is an alternative to complex, almost universal (and therefore expensive to build and administer) general-purpose authorization engines. This alternative is lightweight, simple to construct, and provides an inexpensive way to run authorization modules, each of which is dedicated to evaluating very specific subset of authorization rules. The decisions from these modules are combined with yet other lightweight specialized modules. As a result, for every distinct authorization policy, a specialized version of the authorization engine is composed out of such modules.

What are the benefit(s), if any, of avoiding general-purpose authorization languages and engines for run-time decisions? We can identify several. To start, no matter how completely a language is supported by an authorization engine, there will always a case that it does not support. Even though most modern authorization languages and engines come with extension points, we are not aware of any instance that would enable simple and efficient synthesis of authorization run-time logic out of existing and new logic.

Composing run-time authorization logic from lightweight specialized modules also enables "pay-for-what-you-use" implementations. The run-time and the administrative overheads become proportional to the complexity of the policies enforced and not to the complexity of all the possible policies supported. Last, but not least, the learning curve for administering, as well as the effort for testing authorization logic composed out of simple modules is again believed to be proportional to the complexity of the enforced policy. By avoiding large generic decision engines and replacing them with the architectures and tools for composing customized engines, developers can better meet the goals for short times to market and for developing solutions useful in a wide range of application domains.

Our approach is not in conflict with the principle of designing a system with security in mind from the beginning. The design of distributed applications still has to take into account the security requirements as well as the capability of the security mechanisms and the underlying middleware technology. For instance, unless each employee record in Example 2 is designed to correspond to a separate distributed object, it would be impossible to allow employees to change their own contact information

[1] By "ascendent web.config" we mean a web.config file located down in the directory hierarchy.

without mixing authorization and application logic. What our approach aims at is reducing the effort required to create and adjust adequate A&A controls in the presence of changes to security policies.

5.1 What About AOSD?

The two points discussed above also apply to the aspect-oriented software development (AOSD) methods. Even though AOSD is mostly about dealing with crosscutting concerns that cannot be cleanly modularized, the question of designing decision logic remains. The approach of Lego™-like building blocks combined with a flexible and extensible base for composition, as well as the means of creating new blocks can be employed for designing whatever parts of the aspect in question that the AOSD techniques and tools are able to decouple from the business logic. This also pertains to the choice between large generic policy engines and those composable from specialized light-weight modules.

It is not surprising that the reverse is also applicable, i.e., the implementations of the architectures like the one presented in this paper could benefit from AOSD techniques. An example can be found in [35] which proposes an AOSD-based approach for improving the flexibility and extensibility of security systems at finer levels of granularity than what OO techniques can offer.

6 Conclusions

In this paper, we presented a flexible and extensible authentication and authorization architecture for protecting ASP.NET Web services. While presenting the architecture, we demonstrate the feasibility and benefits of a) the use of lightweight building blocks along with the means for composing them into specialized solutions as well as adding new blocks with custom logic, and b) composing run-time logic for authorization decisions from small encapsulated units of specialized logic. The architecture has been implemented in an actual security solution.

References

1. Schmidt, D.C. and C. Cleeland, *Applying patterns to develop extensible ORB middleware.* IEEE Communications Magazine, 1999. **37**(4): p. 54-63.
2. Singhai, A., A. Sane, and R.H. Campbell. *Quarterware for middleware*. in *18th International Conference on Distributed Computing Systems*. 1998. Amsterdam, Netherlands: IEEE Computer Society.
3. Wang, Y.-M. and W.-J. Lee. *COMERA: COM extensible remoting architecture*. in *Proceedings of COOTS: 4th USENIX Conference on Object-Oriented Technologies and Systems, 27-30 April 1998*. 1998. Sante Fe, NM, USA: USENIX Assoc.
4. Hamilton, G., M.L. Powell, and J.G. Mitchell, *Subcontract; A flexible base for distributed programming.* Operating Systems Review (ACM): Proceedings of the 14th ACM Symposium on Operating Systems Principles, Dec 5-8 1993, 1993. **27**(5): p. 69-79.

5. Balasubramanian, K., et al. *Towards composable distributed real-time and embedded software*. in *WORDS 2003: 8th International Workshop on Object-oriented Real-Time Dependable Systems, 15-17 Jan. 2003*. 2003. Guadalajara, Mexico: IEEE.
6. Othman, O., C. O'Ryan, and D.C. Schmidt, *Designing an adaptive CORBA load balancing service using TAO.* IEEE Distributed Systems Online, 2001. **2**(4).
7. Nahrstedt, K., et al., *QoS-aware middleware for ubiquitous and heterogeneous environments.* IEEE Communications Magazine, 2001. **39**(11): p. 140-8.
8. Venkatasubramanian, N., *Safe 'composability' of middleware services.* Communications of the ACM, 2002. **45**(6): p. 49-52.
9. Krishnamurthy, S., W.H. Sanders, and M. Cukier, *An Adaptive Quality of Service Aware Middleware for Replicated Services.* IEEE Transactions on Parallel and Distributed Systems, 2003. **14**(11): p. 1112-1125.
10. Crowcroft, J., et al., *Channel islands in a reflective ocean: large-scale event distribution in heterogeneous networks.* IEEE Communications Magazine. **40**(9): p. 112-15.
11. Yang, J. and G.E. Kaiser, *JPernLite: extensible transaction services for the WWW.* IEEE Transactions on Knowledge and Data Engineering, 1999. **11**(4): p. 639-657.
12. Houston, I., et al., *The CORBA Activity Service Framework for supporting extended transactions.* Software - Practice and Experience, 2003. **33**(4): p. 351-73.
13. Gittler, F. and A.C. Hopkins, *The DCE Security Service.* Hewlett-Packard Journal, 1995. **46**(6): p. 41-48.
14. OMG, *CORBAservices: Common Object Services Specification, Security Service Specification v1.8*. 2002, Object Management Group, document formal/2002-03-11.
15. Sun, *Java Authentication and Authorization Service (JAAS)*. 2001, Sun Microsystems.
16. Sun, *Java Authorization Contract for Containers*. 2002.
17. Chapin, S.J., et al., *New model of security for metasystems.* Future Generation Computer Systems, 1999. **15**(5): p. 713-722.
18. Andersen, A., et al. *Security and middleware*. in *WORDS 2003: 8th International Workshop on Object-oriented Real-Time Dependable Systems, 15-17 Jan. 2003*. 2003. Guadalajara, Mexico: IEEE.
19. Beznosov, K., et al. *A Resource Access Decision Service for CORBA-based Distributed Systems*. in *Annual Computer Security Applications Conference*. 1999. Phoenix, Arizona, USA: IEEE Computer Society.
20. OMG, *Resource Access Decision Facility*. 2001, Object Management Group.
21. Jajodia, S., et al., *Flexible support for multiple access control policies.* ACM Transactions on Database Systems, 2001. **26**(2): p. 214-60.
22. Siewe, F., A. Cau, and H. Zedan. *A compositional framework for access control policies enforcement*. in *Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering, FMSE'03, Oct 30 2003*. 2003. Washington, DC, United States: Association for Computing Machinery.
23. Microsoft, *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication*. 2002: Microsoft Press.
24. Hartman, B., et al., *Mastering Web Services Security*. 1st ed. 2003, New York: John Wiley & Sons, Inc.
25. Beznosov, K. *Protecting ASP.NET Web Services: Experience Report*. in *preparation*. 2004.
26. Microsoft, *Altering the SOAP Message Using SOAP Extensions*. 2002.
27. Fleury, M. and F. Reverbel. *The JBoss extensible server*. in *ACM/IFIP/USENIX International Middleware Conference*. 2003. Rio de Janeiro, Brazil: Springer-Verlag.

28. Wang, N., et al., *Evaluating meta-programming mechanisms for ORB middleware*, in *IEEE Communications Magazine*. 2001. p. 102-113.
29. Baldoni, R., C. Marchetti, and L. Verde, *CORBA request portable interceptors: analysis and applications.* Concurrency and Computation Practice & Experience, 2003. **15**(6): p. 551-579.
30. Narasimhan, N., L.E. Moser, and P.M. Melliar-Smith, *Interceptors for Java Remote Method Invocation.* Concurrency Computation Practice and Experience, 2001. **13**(8-9): p. 755-774.
31. Barkley, J., K. Beznosov, and J. Uppal. *Supporting Relationships in Access Control Using Role Based Access Control.* in *Fourth ACM Role-based Access Control Workshop*. 1999. Fairfax, Virginia, USA.
32. Sandhu, R., et al., *Role-Based Access Control Models.* IEEE Computer, 1996. **29**(2): p. 38-47.
33. Beznosov, K. *Object Security Attributes: Enabling Application-specific Access Control in Middleware*. in *4th International Symposium on Distributed Objects & Applications (DOA)*. 2002. Irvine, California, USA: Springer-Verlag.
34. OMG, *Security Domain Membership Management Service, Final Submission*. 2001, Object Management Group.
35. Gao, S., et al. *Applying Aspect-Orientation in Designing Security Systems: A Case Study.* in *The Sixteenth International Conference on Software Engineering and Knowledge Engineering*. 2004. Banff, Alberta, Canada.

Appendix A. Overview of Resource Access Decision Architecture

With the RAD architecture, an application requests an authorization decision from a RAD authorization service and enforces the decision. A RAD service is composed of the following components (Figure 7): The *AccessDecisionObject* (ADO) 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 certain 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 potentially several 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 resource associated with the provided resource name.

Figure 7 shows interactions among components of authorization service:

1. The authorization service receives a request via the ADO interface.
2. The ADO obtains object references to those *PEs* associated with the resource name in question and an object reference for the responsible *DC*.
3. The ADO obtains dynamic attributes of the principal (client) in the context of the resource name and the intended access operation.

4. The ADO delegates an instance of *DC* for polling the *PEs* (selected in Step 2).
5. The *DC* obtains decisions from *PEs* and combines them according to its policy.
6. The decision is forwarded to the ADO, which returns it to the application.

Further details on RAD architecture could be found in [19, 20].

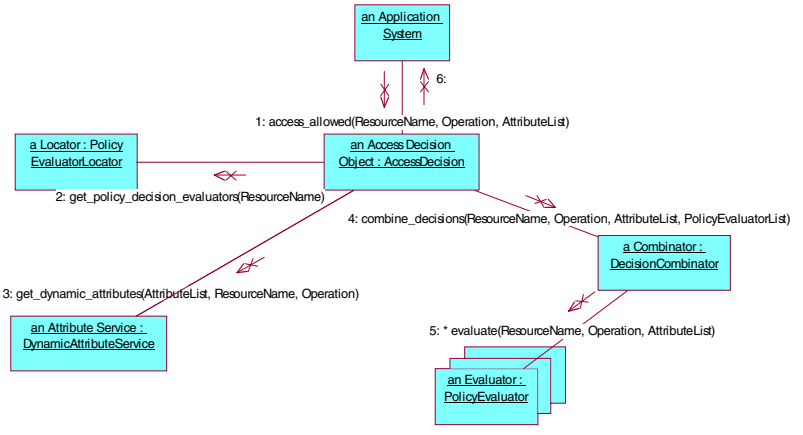

Fig. 7. RAD interaction diagram

Appendix B. Overview of Attribute Function Architecture

The concept of the Attribute Function (AF), as an addition to the traditional decision and enforcement functions, has been proposed in [33]. Its application to CORBA was developed as well [34].

AF has simple syntax: it accepts (middleware-specific) data that are necessary for identifying the state of the target object and returns a set of application-specific attributes of that object. The target object state is necessary for retrieving such object metadata. Since the semantics of object attributes is very specific to the application being protected, AF is provided by the application and not by the middleware or security layers.

The introduction of the AF in the security mechanism design for distributed applications is expected to enable the use of application-specific factors in security policy decisions without coupling enforcement and decision functions with the application.