Experience Report: Design and Implementation of a Component-Based Protection Architecture for ASP.NET Web Services

Konstantin Beznosov

Laboratory for Education and Research in Secure Systems Engineering,
University of British Columbia
beznosov@ece.ubc.ca

Abstract. This report reflects, from a software engineering perspective, on the experience of designing and implementing protection mechanisms for ASP.NET Web services. The limitations of Microsoft ASP.NET container security mechanisms render them inadequate for hosting enterprise-scale applications that have to be protected according to diverse and/or complex application-specific security policies. In this paper we report on our experience of designing and implementing a component-based architecture for protecting enterprise-grade Web service applications hosted by ASP.NET. Due to its flexibility and extensibility, this architecture enables the integration of ASP.NET into the organizational security infrastructure with less effort by Web service developers. The architecture has been implemented in a real-world security solution. This paper also contributes a best practice on constructing flexible and extensible authentication and authorization logic for Web services by using Resource Access Decision and Attribute Function (AF) architectural styles. Furthermore, the lessons learned from our design and implementation experiences are discussed throughout the paper.

1 Introduction

ASP.NET container is a popular hosting environment for Web services built and run atop Microsoft Windows and .NET platforms. However, the ASP.NET security architecture [11, 13], as provided "out-of-the-box," is not sufficiently scalable, flexible, and easily extensible to be adequate for enterprise applications [3]. As we describe in [8], ASP.NET supports limited authentication and group/user-based authorization, both bound to Microsoft proprietary technologies (Windows domains and Passport [12]). If a Web service application needs to be protected via third-party authentication or authorization services available in the enterprise security infrastructure, the real-world developers have two options. The first is to develop homegrown container security extensions, which are hard for average application developers to get right. The second is to program the security logic into the Web service business logic, making the resulting application costly to change 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 hard-to-avoid security-related bugs. Because our architecture achieves fine-grain flexible decomposition of the security logic into components, the design allows a higher degree of security logic reuse whilst supporting application-specific security policies and the separation between business and security logic. We expect the reuse will lead to fewer errors by developers.

Due to its flexibility and extensibility, our component-based protection architecture enables the integration of ASP.NET into the organizational security infrastructure with less effort by Web service developers. The architecture is flexible because it allows for the configuring of machine-wide authentication and authorization functions, and for their overriding for a sub-tree of the Web services (up to an individual application) in the directory-based ASP.NET hierarchy. Its extensibility is revealed through the support of a wide variety of authentication and authorization (A&A) logic, as long as the logic can be translated into a .NET component and/or accessed (possibly via a proxy) through a predefined .NET API. Furthermore, one can reuse those components by combining authorization decisions from them according to predefined or customized rules.

These properties were achieved via

1. separating custom SOAP [21] extension modules, which act as ASP.NET-specific A&A enforcement logic, from the A&A decision logic;
2. following Resource Access Decision (RAD) architecture style [4, 5, 17], which, through the decomposition of the authorization engine into components, makes the customization of access control decision logic easier and avoids the need for a generic policy evaluation engine;
3. taking advantage of the extensibility, inheritance, and caching features of ASP.NET *web.config* configuration mechanism; and
4. separating the logic of retrieving attributes from the authorization and business logic by following the Attribute Function (AF) approach [2].

Although this paper discusses the design of a protection architecture for Web services, we believe that our approach and design decisions could be useful in a broader context of component-based protection sub-systems for distributed applications.

This paper is organized as follows. The next section reviews ASP.NET Web services. Section 3 discusses the requirements for the design. Intertwined with the discussion of the design decisions and lessons learned, an architecture description follows in Section 4. To illustrate the architecture capabilities, we present two examples of policies and corresponding configurations in Section 5. We conclude in Section 6.

2 Overview of ASP.NET Web Services

This section provides background information on ASP.NET Web services technology for the uninformed reader to aid with understanding the rest of the paper. Those familiar with the technology can skip to Section 3.

A Web service is an XML-based messaging interface to computing resources that is accessible via Internet protocols. A Web service front end can be added to an existing information-processing infrastructure. Alternately, applications can be engineered to use a consistent Web services model in all tiers, from data stores and back-ends to middle and presentation tiers. A key Web services technology, SOAP [21] is a unidirectional XML-based protocol for passing structured information.

ASP.NET is the most popular platform among Microsoft technologies for engineering Web services. ASP.NET Web Services rely upon ASP.NET, .NET Framework, IIS, and, underneath them all, the Windows OS platform. ASP.NET can be viewed as a middleware container, similar to J2EE, for hosting components of .NET-based distributed applications accessible via Microsoft's Internet Information Server (IIS). Since ASP.NET runs in .NET's virtual machine, common language runtime (CLR)—whereas IIS is a regular Windows executable—ASP.NET_ISAPI dynamically linked library (DLL) acts as a bridge between the two, as shown in Figure **1**. The DLL receives HTTP requests for URLs ending with specific extensions, the one for ASP.NET Web Services being .asmx.

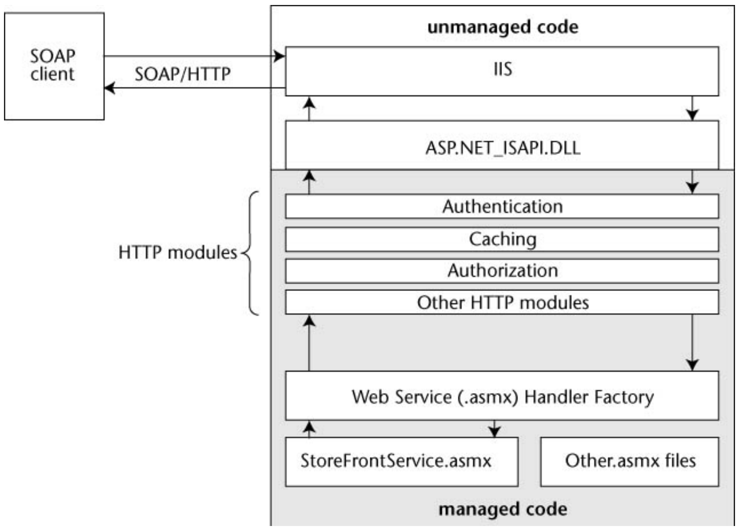

Fig. 1. Request handling by ASP.NET Web services

While running in unmanaged code, IIS forwards a request to the ASP.NET_ISAPL.DLL first. The DLL passes the request to ASP.NET, where the request passes through registered HTTP modules acting as invocation interceptors and reaches the Web Service Handler Factory. The factory uses the information in the URL to determine which Web service implementation should handle the request. ASP.NET dispatches the request to the implementation on demand of the factory. Not used only for performing security functions, the HTTP modules are also used for protecting ASP.NET Web services "out-of-the-box."

Discussed in detail in [8], the ASP.NET Web services preinstalled security mechanisms consist of the security available for the building blocks of these services and SOAP security. Overall, Microsoft products provide a convenient family of technologies to support the security of modest-sized applications with little effort. However, when the security requirements reach the enterprise scale, one needs either significant amounts of in-house development or additional third-party products and services to fill the gap. Fortunately, .NET in general and ASP.NET in particular have architectures that accommodate various security extensions. We designed our protection architecture as an extension to a typical ASP.NET installation.

3 Requirements

The design of the architecture was driven by its requirements and the underlying technology, ASP.NET. The functional objectives of the architecture were to allow flexible authentication and authorization for ASP.NET Web service applications. It was required to support “out-of-the-box” the following types of data (a.k.a. security tokens) for client and message authentication:

- user name and password from the HTTP header, a.k.a. HTTP Basic Authentication (HTTP-BA),
- ASP.NET Session state object with a pre-configured name,
- “stringified” credentials token found in any of the following:
- the custom field of the HTTP header, and/or
- HTTP cookie with a pre-configured name, and/or
- header block of the SOAP message that carries the request to the ASP.NET Web service, similar to WS-Security [15] (WSS).

One of the lessons learned from the requirements engineering exercise was that the end users did not care about the compliance with the security standards related to Web services as long as the design was in the spirit of those standards and therefore enabled eventual compliance with them in the future. The likely reason was the lack of plans for mixing heterogeneous (i.e., produced by different development teams) Web services. That is, no cross-enterprise Web service deployments were envisioned.

Another conclusion drawn from the work concerns the difficulty of determining a practical set of compliance criteria. Taking into account the flexibility of the information architectures for WSS and related specifications, we found it hard to define what a compliant implementation is expected to do. Furthermore, without prior agreement between application owners about the WSS profiles, two compliant applications would not necessarily interoperate in a useful manner.

In regards to authorization, the architecture was required to support a) third-party enterprise-wide A&A products, such as Policy Director [9], SiteMinder [14], getAccess [7], etc., b) selective availability of some service methods for public (i.e., anonymous) access, and c) simple variations of authorization logic.

The architecture was also required to be extensible enough to accommodate new types of A&A logic, e.g., access restriction based on the IP addresses of the Web service clients or the access day and time. Since it was impossible to envision all probable instances of A&A policies, the extension mechanisms had to be sufficiently generic. We also anticipated the need to compose new authorization policies out of existing ones (where developers could re-use much of the existing A&A logic).[1] Because this paper focuses on A&A, we do not discuss other requirements such as audit.

[1] For example, some publicly accessible methods with the remaining methods controlled by the enterprise-wide authorization.

4 Architecture Overview

To integrate with ASP.NET run-time, the architecture takes advantage of the ASP.NET generic interception mechanism, *SOAPExtension* [10], intended for additional processing of SOAP messages. As shown in Figure **2**, our custom version of SOAPExtension (labeled "interceptor") performs the 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.

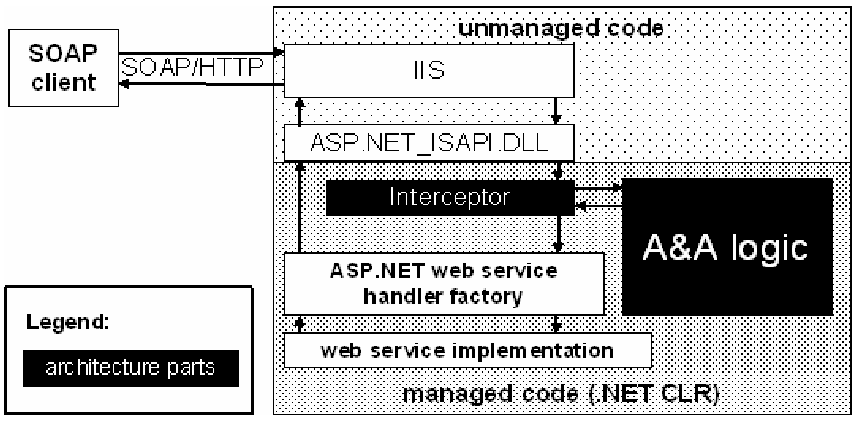

Fig. 2. General organization of the architecture into an interceptor and A&A logic

Since the purpose of the architecture is A&A, SOAP messages are processed on their way in and only after ASP.NET run-time has successfully parsed all SOAP-specific XML and HTTP formatting. If the protection of data in transit were a requirement, then the additional processing of the SOAP messages on their way out would be necessary. Since we did not anticipate it to undergo future changes, the interceptor has been designed with no extension or modification points. On the other hand, its design and implementation were optimized for performance since it was the most frequently used component in the architecture, invoked each time a protected Web service is accessed. Design for change [19], however, was a major goal for the decision part of the architecture, labeled as "A&A logic" in Figure **2**. This part is composed of several other elements as described in the following sections.

4.1 Authentication

Authentication is commonly divided into two phases: retrieving authentication data and validating it. Following the same division, our *CredentialsRetriever* objects specialize in retrieving authentication data. Each retriever implementation is responsible for extracting particular data types (e.g., user name and password encoded as HTTP-BA, credentials token found in the SOAP message header, etc.) from the appropriate locations and encapsulating them in *Credential* objects. In the design of the authentication-related components, we wanted to isolate anticipated changes due to variations in authentication policies ("What data is acceptable for authentication?") from the rest of the architecture. For this purpose, retrieved authentication data and retrievers themselves are represented as implementations of *Credential* and *CredentialsRetreiver* interfaces. This approach allows for adding new modules of retrieving logic to the architecture by application developers, owners, or third-party vendors. For instance, the use of new types of authentication data, such as a client's public key certificate in requests over HTTPS, could be accommodated by developers by creating a new implementation of *CredentialsRetreiver* that retrieves the corresponding attributes of the HTTPS connection and packages them into a new instance of *Credential*. Before retrieved credentials can be used in authorization decisions, they need to be validated.

There are several reasons why credentials validation is separated from the retrieval phase and delayed until authorization. First, some credentials could be computationally expensive to validate. For instance, the validation of credentials signed by a private key requires public key operations as well as potentially unbound delays due to the checking certificate revocation lists. Second, only during the authorization step is it determined which credentials will be used for authorization. For example, if a request is accompanied by a certificate and a username-password but only the certificate is used, then there is no need to validate the latter. Third, some useful policies might call for the evaluation of the same credential with more than one authentication authority. Yet the fourth reason is due to the frequent co-location of authentication and authorization services in enterprise security servers. Lumping authentication and authorization steps in one batch and sending it to a remote server allows for a substantial reduction of the communication overhead in such cases.

Delaying credentials validation until the authorization phase, however, turns out to have a disadvantage as well. Those authorization components that implement the *PolicyEvaluator* interface have to contain credential-specific validation logic. In retrospect, a better design could be to encapsulate such validation logic into objects as parts of the credentials and configure binding between validators with the credentials.

4.2 Authorization

An authorization decision is reached in a three-step process, which is supported by the structure based on RAD and AF architectural styles. Initial decisions are made by zero or more predefined or custom authorization modules referred to as *PolicyEvaluator* (PE). The simplest PE is one that always returns the same decision, e.g., "deny," "permit," depending on its static configuration. Clearly, it ignores any credentials or other attributes of the request or target in question, environment, or the history about the previous requests. Despite its dullness, such a PE turns out to be very handy for testing, debugging, and deploying Web service applications and the architecture itself. More interesting PEs, supplied with the architecture implementation, grant access based on the IP address of the request sender, the name of the Web service target and its methods, and the decisions provided by an enterprise authorization server. The strength of RAD architectural style is in the support of fairly sophisticated authorization policies (see [1] for an example) without the need for complex authorization engines. This support is achieved by combining run-time decisions from several simple PEs into one at the second step, performed by a *DecisionCombinator* (DC).

Another reason for dividing the authorization process into the phases of evaluating (possibly several) policies and combining evaluation results, i.e., decisions, is to enable a high degree of authorization components reuse. Based on prior experience with protection for enterprise applications, we expected that, on the one hand, authorization policies would vary not only from organization to organization but also from application to application. On the other hand, common elements of authorization logic (e.g., decisions based on the roles, groups, and other attributes of the users) recur in most policies, making them perfect candidates for reusable components.

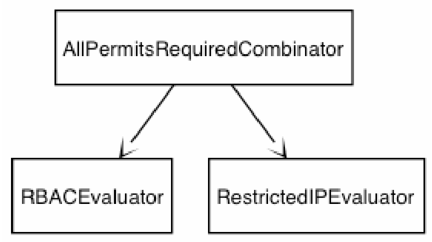

Fig. 3. Resulting configuration with the PE restricting access based on the sender's IP address

To appreciate the power of DC&PEs approach, consider a composition of "All Permits Required" DC with a role-based access control (RBAC) [20] PE. They implement authorization based on user roles and their hierarchies. If the application owners decide to restrict access further to a particular range of IP addresses, they can do so by adding a PE that authorizes IP addresses, instead of modifying the fairly complex logic of the RBAC PE. The result is shown in Figure 3. Support for policies in which PEs might have different priorities is enabled through the use of (unique) PE names so that custom DC logic can discriminate between them.

The authorization process continues to its third stage. This stage is important for achieving *fail-safe defaults* in those cases when a DC experiences a failure due to a design or implementation error and does not come to a binary decision. During this stage, the interceptor renders any decision, except "permit," received from the DC to "deny" and thus reaches 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: name and attributes of the Web service, its policy domain, and the method to be invoked. All this information is constructed into a *permission*. Thus, the authorization process results in a decision on whether a given permission should be granted to a given subject (represented by its credentials). If so, 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 elements, as shown in Figure 4.

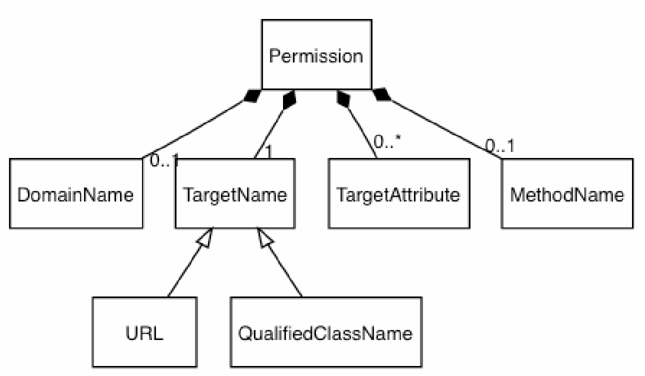

Fig. 4. Elements of the permission generated by the default permission factory

1. *TargetName*—the name of the target Web service can be represented by either a URL or the .NET class name of the service implementation. The URL represents the web server's interpretation of the URL from the corresponding HTTP request. The use of URLs for naming Web services is less attractive in ASP.NET because the same .NET class can be reused to create separate instances of Web services. In the ASP.NET environment, a single file hosts each target Web service. Different URLs can be used to invoke the same implementation class. The presence of these synonyms can pose a challenge to the security administrator's primary goal—maintaining proper security policy for Web services. The use of the .NET class name instead of the URL means that all instances of a Web service application can share the same authorization policy rules. This reduces the cost of maintenance and allows the same application logic to be protected no matter how many names under which it is deployed. When used together with the domain capability, several instances of the same Web service can be treated the same, or distinctly, as appropriate for the application structure.
2. *DomainName*—the use of a domain classifier is borrowed from CORBA Security [6, 16] architecture, whose policy domains support different security requirements for implementations of the same interfaces. In our architecture, optional domains allow discrimination between those same implementations of a Web service that have different access control requirements. Another purpose of domains is to allow for a logical grouping of several Web services, perhaps so that they can share an authorization server or its policy database. Since the means of determining the domain of a Web service is highly specific to the application and its authorization policies, our architecture provides a simple version of a domain retriever and a means for replacing it with custom implementations.
3. *Target attributes*—further differentiation among Web service instances is achieved through an optional list of one or more name-value pairs holding target attributes. For example, a Web service representing a bank account manager could have at-

tributes that identify the branch to which all the managed accounts belong, provided the division of the accounts among such managers is based on the branches. As it was argued in [2], 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 at run time are directly based on the prior work on Attribute Function [2, 18]. The extensible retrieval mechanism is designed as a replaceable *TargetAttributeRetriever* interface, with a simple implementation provided by the architecture implementation.

4. *Method*—since ASP.NET, at the time of this work, supported only RPC semantics for interactions with hosted Web service implementations, acceptable SOAP messages had to specify the method of the .NET implementation class responsible for processing the request. As with other RPC-based middleware technologies, it was important to support these authorization decisions based on method name. The method name is optional in the constructed permission to support types of applications that do not require authorization policy granularity at the method level.

Table 1 shows examples of permissions:

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

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. The configuration, described below, determines which permission factory, DC&PEs, credential retrievers, and other replaceable parts of the architecture are used for serving requests for each Web service instance.

4.4 Replaceable Parts

As stated before, the flexibility and extensibility of the architecture is achieved via designing most of its elements to be replaceable. Any of the black boxes in Figure 5 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 [6, 16], 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 (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. Furthermore, 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, COM+, or EJB implementations. Flexible and manageable configuration turns out to be critical for making fine-grain and yet scalable replaceability workable.

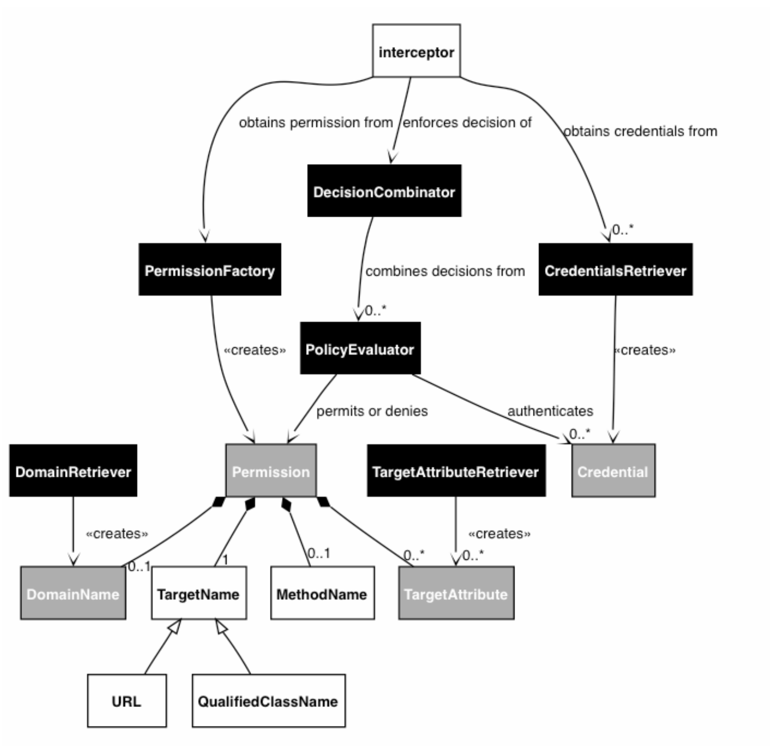

Fig. 5. Key elements of the architecture

4.5 Configuration

Flexible and scalable configuration is critical in order for our architecture to be extensible and, at the same time, carry low administration or run-time overhead. Since an ASP.NET container might host many Web services, each with its own security requirements, and deployment and maintenance life cycles, the run-time changes to the configuration should not result in the restart of the container or its lasting performance degradation. It turns out that ASP.NET configuration architecture, with settings defined in *web.config* files, had most features we were looking for.

The use of simplified XML in web.config files enables a flexible hierarchy of configuration elements, as shown in Figure **6**. By leveraging the web.config ability to delegate the handling of new configuration sections to custom handling logic, we developed a simple hierarchical language to define and configure various elements of the A&A decision logic as well as the protection policies that comprise them.

A protection policy can simply be viewed as a collection of specific credential retrievers, PEs, DC, as well as of attribute and domain retrievers, and permission factory. They are defined in other sections of the configuration and the policy only refers to them by name (and possibly re-configures them), thus enabling reuse.

Since all of these elements are defined independently of the policies and have unique names, they can be referenced by more than one protection policy. A singleton in the scope of a web.config instance, *Governing Policy* (GP) specifies which particular policy is used for controlling access to the Web service in question. Thus, one can prepare and test a protection policy, and perform a quick switch to the new policy by just changing the *name* attribute of a GP, a reference to specific protection policy. Multiple policies can be prepackaged and used to alter the behavior of the protection mechanisms in response, for example, to the changes in the threat level.

The hierarchal nature of web.config parsing semantics enables good scalability without losing a fine level of granularity in the control over sub-sets of (or individual) Web services. The GP defined in the root web.config determines the protection of all those Web services, for which no web.config file between the service and the root directory overrides it. Thus, developers can deploy their Web services, which can be administered by changes to the root web.config file. This approach, though, does not address the question of scalable administration for multiple ASP.NET containers, which is an issue for COM+ and standard EJB containers as well. Similar to product-specific solutions on the EJB market, one could remedy the problem by synchronizing web.config files or their specific sections across multiple containers.

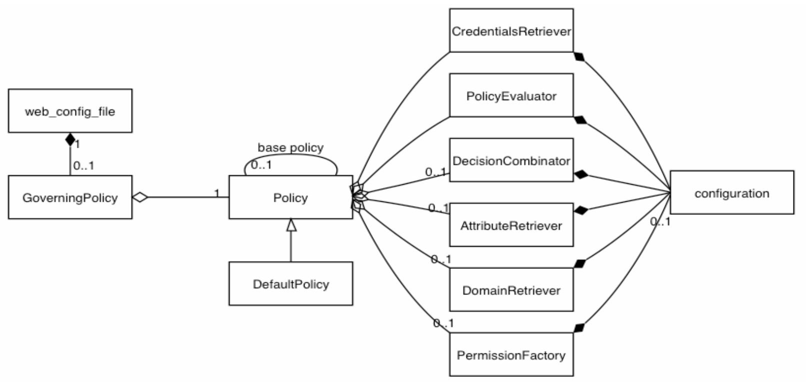

Fig. 6. Simplified model of the configuration elements with default cardinality "0..*"

The configuration flexibility 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 ascendant web.config (i.e., one down in the directory hierarchy). Unfortunately, the freedom of overriding GP means the loss of control over GPs used for protecting the Web services located down in the directory hierarchy. However, this loss can be remedied by the use of OS file system controls, if necessary, by the Windows administrator restricting the rights of other users to modify web.config files down the directory hierarchy. Second, to reduce the effort required for creating policy variations, we also implemented a single inheritance mechanism for protection policy definitions. Thus, a policy could reuse most of the other policy's definition and override just a few elements, such as a *DomainRetriever* or a specific PE.

The performance overhead from storing all configuration information in web.config seems to be relatively small because ASP.NET caches read web.config files and invalidates the cache when the OS detects any changes to the file. Since the behavior or cache of our protection mechanisms is not affected by the changes to descendent web.config files, the goal of isolating Web services that are developed independently but co-hosted by one instance of an ASP.NET container is half-reached. The other half, eliminating the possibility of undesirable effects from changes in the higher levels of the hierarchy, can be achieved by allocating separate directory sub-trees to independent applications and sharing little or no settings through the web.config mechanism. Even though this solution is far from perfect, we believe it is good enough for most environments.

Adding a new component to the protection sub-system requires the simple step of adding an entry with the information about the corresponding .NET assembly, class name, and the name of the component into the web.config file. Afterwards, this component can be referenced in the corresponding sections of web.config. Removing a component involves the same steps but in reverse order. The above steps do not even require stopping the protection sub-system.

5 Examples

To demonstrate the ability of our architecture to be customized through different compositions of replaceable components, we provide hypothetical examples of implementing two different policies. These examples also illustrate the high degree of security logic reuse that, we expect, could reduce the error rate in the corresponding parts of the applications and their supporting infrastructure. Real commercial applications and policies that have used our implementation cannot be discussed due to the lack of permissions from the application owners.

5.1 Example 1: University Course Web Service

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

Policy 1

1. All users should authenticate using user name and password in HTTP header (HTTP-BA).
2. **Anybody** can *look up* course descriptions.
3. **Registration clerks** can *list students* registered for the course and (*un)register* students.
4. The **course instructor** can *list registered students*, *manage course assignments* and *course material.*
5. **Registered** for the course **students** can *get assignments* and *course material*, and *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 **7** with custom-built modules in black.

Configuration 1:

- An HTTP-BA *CredentialRetriever* CR_1 extracts the user name and password from the HTTP request that carried the corresponding SOAP request.

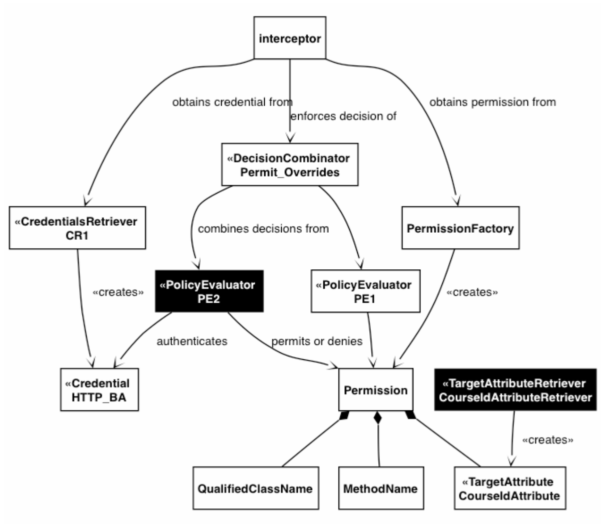

Fig. 7. Configuration 1. Custom-built components appear in the black boxes

- 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. No domain name is used in this configuration. Here is an example: "ca.ubc.CourseMngmnt.SimpleCourse/CourseId=EECE412/GetDescription".
- A pre-built *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.

  User roles and other attributes are retrieved by the PE during or after it validates the credential received from HTTP-BA *CredentialRetriever.* We refrain from discussing this step since it is very specific to the particular student and employee databases used by the university and is irrelevant to the discussion.
- A pre-built *DecisionCombinator* of type *Permit Overrides* grants access if either PE grants access.

This example also illustrates one specific issue with any component-based design: even when each component satisfies its specification, there is no inherent guarantee that the assembled system also does. For instance, PE_2 (which assumes the presence of a *CourseId* attribute in the permission passed to it) depends on the *TargetAttributeRetriever* to retrieve such an attribute and on the *PermissionFactory* to insert the attribute into the permission. All three have to function together for the protection sub-system to function as expected. In our solution, we have not addressed this issue, leaving developers to ensure the consistency of the assembled protection mechanisms manually. The development of a specific automated solution for consistency verification is a potent topic for future research on component-based security subsystems.

5.2 Example 2: Human Resource Web Service for International Organization

Now consider a multinational company that has divisions in Japan, Canada, the USA, 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 for 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* for 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 is used as in Example 1.
- Another *CredentialRetriever* CR_2 obtains an SSL client certificate from the HTTPS connection.
- A pre-built simple *DomainRetriever* always returns the 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. Here is an example: "Japan/com.mega-foo.EmployeeInfo/GetContactInfo".
- Same pre-built *PolicyEvaluator* PE_1 as in Example 1 is used. This time, there are four public methods: FindEmployee, GetEmployeeInformation, GetEmployeeManager, GetSupervisedEmployees.
- A pre-built *PolicyEvaluator* PE_3 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 permits access to any request made by a user with a valid X.509 certificate (retrieved by CR_2) issued by the company.

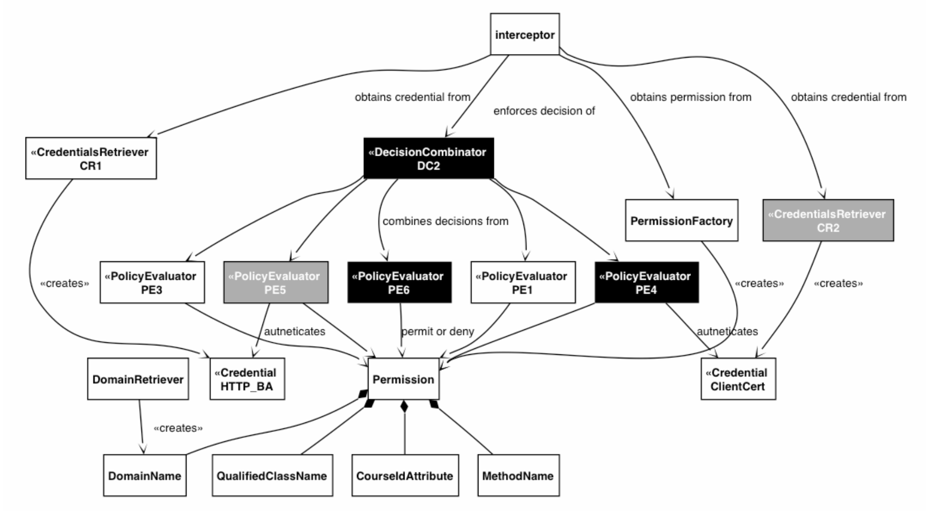

Fig. 8. Configuration 2. Custom-built components appear in black boxes. Generic ones supplied by vendors appear in gray boxes

- A generic RBAC *PolicyEvaluator* PE_5 permits the invocation of different methods based on the role of the user:
  1. Any user with the role "hr employee" can invoke methods that modify contact information and review salary.
  2. Any user with the role "hr manager" can invoke methods permitted to users with role "hr employee" as well as methods that modify an employee's salary, title, and the names of the manager and supervised employees.
- A custom-built *PolicyEvaluator* PE_6 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 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 a valid company 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 for the employee from the same division $(PE_5 \wedge PE_6)$.

The high degree of the architecture composability allows for re-using two pre-built (PE_1 & PE_3) from configuration 1. Even though configuration 2 has three more PEs and one more *CredentialRetriever* than configuration 1, as shown in Figure **8**, 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 with 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 Conclusions and Acknowledgement

This paper reports an experience of designing a flexible and extensible architecture for protecting enterprise-grade ASP.NET Web services. The architecture's flexibility and extensibility have been achieved through a component-based design that follows the architectural styles of RAD [4, 5, 17] and Attribute Function [2]. This architecture has been implemented in a real-world security solution. We described requirements, presented the architecture, and explained the design decisions along with the lessons learned from this work.

The author thanks the anonymous reviewers for their insightful comments that helped to improve this paper. ICICS editorial assistant Ben D'Andrea was instrumental in making this paper more readable.

References

1. Barkley, J., Beznosov, K. and Uppal, J., "Supporting Relationships in Access Control Using Role Based Access Control," in Proceedings of the Fourth ACM Role-based Access Control Workshop, (Fairfax, Virginia, USA, 1999), pp. 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, pp. 693-710.
3. Beznosov, K. Overview of .NET Web Services Security, presented at Distributed Object Computing Security Workshop, Baltimore, MD, USA, 2002.
4. Beznosov, K., Deng, Y., Blakley, B., Burt, C. and Barkley, J., A Resource Access Decision Service for CORBA-based Distributed Systems. in Proceedings of the Annual Computer Security Applications Conference (ACSAC), (Phoenix, Arizona, USA, 1999), pp. 310-319.
5. Beznosov, K., Espinal, L. and Deng, Y., "Performance Considerations for CORBA-based Application Authorization Service," in Proceedings of the Fourth IASTED International Conference Software Engineering and Applications, (Las Vegas, Nevada, USA, 2000).
6. Blakley, B. CORBA Security: an Introduction to Safe Computing with Objects. Addison-Wesley, Reading, MA, 1999.
7. Entrust. getAccess Design and Administration Guide, Encommerce, 1999, 182p.
8. Hartman, B., Flinn, D.J., Beznosov, K. and Kawamoto, S. Mastering Web Services Security. John Wiley & Sons, New York, 2003.
9. Karjoth, G., "The Authorization Service of Tivoli Policy Director," in Proceedings ACSAC, (New Orleans, Louisiana, 2001), pp.319-328.
10. Microsoft. "Altering the SOAP Message Using SOAP Extensions," 2002.
11. Microsoft Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication. Microsoft Press, 2002.
12. Microsoft. Microsoft .NET Passport, 2001.
13. Microsoft. "Securing XML Web Services Created Using ASP.NET" in *.NET Framework Developer's Guide*, 2001.
14. Netegrity. SiteMinder Concepts Guide, Waltham, MA, 2000, 78p.
15. OASIS. Web Services Security: SOAP Message Security 1.0 (WS-Security 2004), 2004.
16. OMG. CORBAservices: Security Service Specification v1.7, formal/01-03-08, 2001.
17. OMG. Resource Access Decision Facility, formal/2001-04-01, 2001.
18. OMG. Security Domain Membership Management Service, Final Submission, 2001.
19. Parnas, D.L. "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, SE-5(2):128-137, 1979.
20. Sandhu, R. et al. "Role-Based Access Control Models," IEEE Computer, 29(2):38-47,1996.
21. W3C. SOAP Version 1.2 Part 1: Messaging Framework, W3C, 2002.