

# Security Services in Mainstream Enterprise-Oriented Middleware Platforms

*Tom Goovaerts*

*Bart De Win*

*Wouter Joosen*

*Report CW 406, March 2005*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Security Services in Mainstream Enterprise-Oriented Middleware Platforms

*Tom Goovaerts*

*Bart De Win*

*Wouter Joosen*

*Report CW 406, March 2005*

Department of Computer Science, K.U.Leuven

## **Abstract**

Security is an essential requirement of web-based enterprise applications. This report overviews the state of the art in security services of today's commercial middleware platforms typically used to build this family of applications. The overview includes security services of large scale server-side middleware platforms, including web services, as well as their small footprint counterparts for mobile devices.

**Keywords :** security, middleware

**CR Subject Classification :** C.2.4, K.6.5

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Security Services in Middleware for Enterprise Applications</b>	<b>5</b>
2.1	Introduction . . . . .	5
2.2	J2EE . . . . .	5
2.2.1	Introduction . . . . .	5
2.2.2	Authentication . . . . .	7
2.2.3	Access Control . . . . .	9
2.2.4	Message Security . . . . .	12
2.2.5	Audit . . . . .	12
2.3	CORBA . . . . .	13
2.3.1	Introduction . . . . .	13
2.3.2	Authentication . . . . .	14
2.3.3	Access Control . . . . .	16
2.3.4	Message Security . . . . .	18
2.3.5	Non repudiation . . . . .	19
2.3.6	Audit . . . . .	19
2.3.7	CORBA Component Model . . . . .	20
2.4	Microsoft .NET Framework . . . . .	21
2.4.1	Introduction . . . . .	21
2.4.2	Authentication . . . . .	21
2.4.3	Access Control . . . . .	23
2.4.4	Message Security . . . . .	25
2.4.5	Audit . . . . .	25
2.5	Web Services Security . . . . .	25
2.5.1	Introduction . . . . .	25
2.5.2	WS-* family of security standards . . . . .	26
2.5.3	SAML . . . . .	28
2.5.4	Other web service-related security standards . . . . .	31
<b>3</b>	<b>Security in Middleware for Mobile Devices</b>	<b>32</b>
3.1	Introduction . . . . .	32
3.2	Operating system . . . . .	32
3.2.1	Device access . . . . .	33
3.2.2	Cryptographic Libraries . . . . .	33
3.2.3	Data Security . . . . .	33
3.2.4	Message security . . . . .	33
3.3	J2ME . . . . .	33

3.3.1	Protection Domains . . . . .	34
3.3.2	Application Signing . . . . .	34
3.3.3	Communications Security . . . . .	35
3.4	Microsoft .NET Compact Framework . . . . .	35
3.4.1	Authentication . . . . .	35
3.4.2	Access Control . . . . .	35
3.4.3	Data and Communication Security . . . . .	36
3.5	CORBA . . . . .	36
3.5.1	minimumCORBA . . . . .	36
3.5.2	Lightweight CORBA Component Model . . . . .	36

# List of Figures

2.1	The J2EE architecture . . . . .	6
2.2	An overview of the different authentication mechanisms in J2EE. . . . .	7
2.3	An overview of the J2EE role-based access control mechanism. . . . .	10
2.4	The CORBA architecture and the CORBA Security Service. . . . .	14
2.5	A secure association and its policies. . . . .	15
2.6	An example of how access control decisions are made in CORBASec. . . . .	18
2.7	The architecture of a multitiered .NET application. . . . .	21
2.8	A web services infrastructure and the position of the WS-Security and SAML standards. . . . .	26
2.9	An example of a SAML interaction that uses authentication and authorization assertions. . . . .	29

# Chapter 1

## Introduction

This report discusses the security services in middleware for web-based enterprise applications. The middleware that is specific for this kind of applications can be divided in two classes: large scale server-side middleware and client-side middleware for mobile devices.

Section 2 discusses security services in server-side middleware systems for web-based enterprise applications. First, the security services of the major commercial middleware platforms J2EE, .NET and CORBA are described. Then, an overview of the most important security-related standards and specifications of the web services world is given. These standards and specifications are not middleware by themselves, but they are to be implemented by middleware products that support web services.

Section 3 discusses the security functionality that is present on mobile devices. Enterprise applications that have client-side components on a mobile device demand security functionality from the platforms that are used on the mobile devices. Since mobile devices have limited resources and middleware security often relies on functionality offered by the underlying operating system, first an overview is given of the security features found in the most common mobile operating systems: Palm OS and Windows CE. Then, the security services in the available mobile platforms for enterprise applications are described. The platforms that are discussed are J2ME, the .NET Compact Framework and lightweight CORBA implementations.

## Chapter 2

# Security Services in Middleware for Enterprise Applications

### 2.1 Introduction

Enterprise applications currently make extensive use of middleware. The middleware layer diminishes the complexity that arises with the distribution of large-scale software systems. The current trend in distributed applications is to move from the two-tiered client-server model towards three-tiered architectures to support web-based clients. Several middleware platforms exist that support these architectures. Since security is of great importance for this class of applications, these middleware platforms have to be able to meet their security requirements.

This chapter gives an overview of the security services in today's important middleware platforms. Section 2.2 discusses security in Sun's J2EE, section 2.3 describes the CORBA Security Service and section 2.4 handles Microsoft's .NET Framework. Section 2.5 discusses security standards for web services.

### 2.2 J2EE

#### 2.2.1 Introduction

Sun's Java 2 Enterprise Edition (J2EE) specification [26] defines a collection of services and component frameworks for building multi-tiered enterprise applications with the Java language and technology. A J2EE application can contain four kinds of components: applets and application clients on the client tier, web components (Servlets and JSP's [27]) on the web tier and Enterprise JavaBeans [25] on the business tier. These components execute in a *container*, a piece of middleware that provides various J2EE services (such as security) and hides complex aspects that arise in distributed applications. J2EE components may only interact with each other through their containers. Each component type has a corresponding container type, so the J2EE containers are: the applet

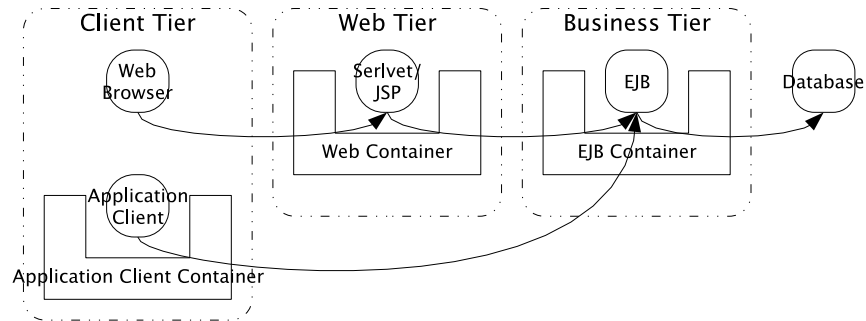


Figure 2.1: The J2EE architecture

container (typically a web browser), the application client container, the web component container and the enterprise bean container. This document mainly discusses security at the server-side containers: the web container and the EJB container. Figure 2.2.1 shows the various J2EE components and containers and their interactions.

The J2EE specification defines a security architecture for containers. Security for components is provided by the container they run in and can be either *declarative* or *programmatic*. The former kind is transparent to the code of the component and is specified in the component's configuration files, the latter consists of explicit API calls in the component's code. Declarative security is preferred by the J2EE specification because it promotes components that only implement business logic. J2EE uses a number of abstractions throughout its security architecture.

**Principal** A principal represents an entity that can be authenticated. Web and EJB containers typically have their own principal whereas application client containers operate under the user's principal.

**Security Attributes** Each principal has a set of security attributes, that can serve many purposes. A security attribute holds some security-related information that can be used by various security mechanisms. Examples of a security attribute are a X.509 certificate for authentication or a set of permissions for access control.

**Security (Policy) Domain or Realm** This is a scope of resources over which some security policy is applied.

The J2EE specification supports several security requirements[26, 25, 27, 12, 8]: authentication, access control, message security and a limited form of auditing. The subsequent sections will discuss each of them in detail.

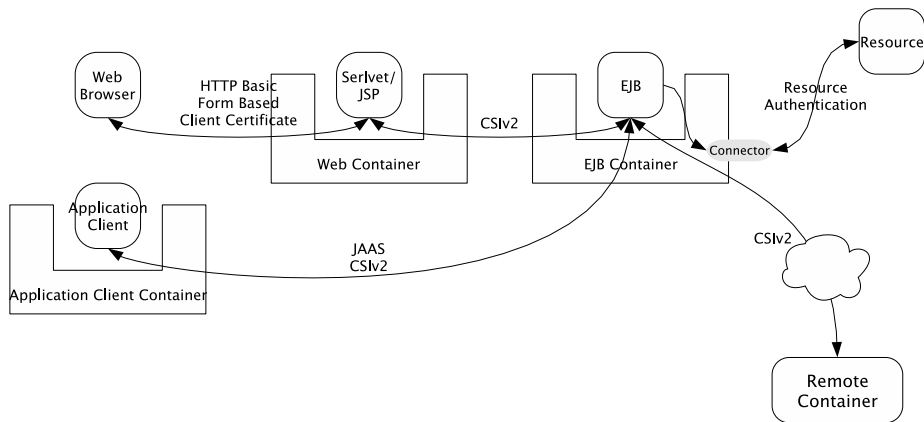


Figure 2.2: An overview of the different authentication mechanisms in J2EE.

## 2.2.2 Authentication

J2EE uses the notion *principal* for an entity that can be authenticated. A principal that authenticates itself by some authentication protocol is identified by a *name* and uses *authentication data* for authenticating itself. The result of the authentication process is the association of *credentials* with the principal. A principal's credentials is the set of its security attributes that can be used to authenticate the principal. A password or a Kerberos ticket, for example, are credentials. Figure 2.2 gives an overview of the different J2EE authentication mechanisms that are discussed.

**User Authentication** The authentication protocol for a user is performed between the piece of software that can control the user interface of the application and the entry point of the application in the enterprise. The client can communicate through a browser with a web container or he can use an application client container that interacts directly with the EJB container.

Most often, users interact with the application through a web browser that communicates with the web container of the application through the HTTP protocol. Consequently, client authentication in this case has to be applied over HTTP and uses the client's web browser as the interaction point with the user. The J2EE specification dictates a number of authentication protocols that a web container must support for authenticating clients through their web browser:

**Basic HTTP** The first option for web browser authentication is the basic username/password scheme. Username and password are transmitted in clear-text from the client to the web container.

**Form Based** The login form can be customized for accepting other kinds of authentication data or just for changing the layout, but this mechanism also cannot transport the user's credentials to the web container in a secure way.

**Client Certificate** This makes use of the HTTPS (HTTP over SSL) protocol to authenticate the client to the application. This mechanism is strong but it requires the user to have its public key certificate, which is currently not very common.

The second way the user can communicate with the application is through an Application Client. Application Clients can easily provide custom authentication mechanisms because they control all interaction with the user. Application client containers are encouraged to use JAAS [14], the java authentication and authorization service, but no explicit authentication requirements are made in the specification. Authentication at the application client must be hardcoded in the application client component, whereas authentication at the web container is fully transparent.

**Identity Propagation** Components of a J2EE application typically don't reside on the same host but are distributed. Therefore, a J2EE component can be configured at deployment to either run under the caller's identity or run under another identity specified at deployment.<sup>1</sup> In the first case the next component in the call chain sees the original client's identity as the identity of the calling principal. In J2EE, this form of identity propagation is called *propagated caller identity*. In the second case, called *run-as identity*, the deployer chooses another identity that must be passed on to other components. By default, propagated caller identity is used. In both cases, only the identity of the caller is passed, not its credentials.

These identity propagation mechanisms are insecure because there is no way to know if the propagated identity is really the original caller's identity. A container in between might have called the next container under an identity that has more privileges than the client to perform some malicious action. Therefore, identity propagation is used only in the controlled enterprise environment where communicating components can already trust each other.

**Remote Container Authentication** It is also possible to establish an explicit trust relation between EJB containers and other J2EE containers. For example, this would be the case when two EJB's want to communicate over the internet. When two such components want to interoperate, they make their invocations by means of the CORBA IIOP protocol [7]<sup>2</sup>. These IIOP invocations can be secured using SSL/TLS, allowing containers to authenticate to each other prior to making invocations.

**Resource Authentication** J2EE components can use external *resources* (typically Enterprise Information Services) that are not in a container. An example of such a resource is a database. Resources are accessed in J2EE through a *connector*, that sets up a connection with the resource and allows components to interact with it. Resources often have their own authentication mechanisms that are incompatible with the J2EE authentication model. Therefore, J2EE connectors have to be able to authenticate to the resource using resource-specific authentication data. The configuration of the authentication data can be either

---

<sup>1</sup>Principals are identified by their name.

<sup>2</sup>The integration of RMI and IIOP is specified in the RMI-IIOP protocol [6].

container-managed or component-managed. Components state which approach they use in their deployment descriptor.

- The container-managed approach is called *configured identity*. This means that the container administrator configures the container to use preconfigured authentication data for each resource. When the component sets up a connection to the resource, the container will ensure that the configured authentication data are passed to the resource. This way, the authentication is transparent for the component.
- When configured identity is not flexible enough, the component can also programmatically authenticate to the resource when making a connection using its own customized authentication data. This component-managed approach is called *programmatic authentication*.

Besides these two mechanisms, some other mechanisms are mentioned but not required. *Principal Mapping* means that the identity to be used to authenticate to the resource depends on the security attributes of the calling principal. *Credential mapping* is an extension of this mechanisms that allows a mapping from the caller principal and security attributes to a completely different form of authentication data. *Caller impersonation* means that the principal and credentials of the caller are used when authenticating to the resource.

### 2.2.3 Access Control

The access control model of J2EE is based on the role-based access control model by E. Sandhu [36]. The policy can be specified declaratively or can be implemented in the code of a component by means of a number of API calls. Access control is provided by web containers and EJB containers. The underlying model of these two access control mechanisms is identical so it will be discussed first in general.

**General Model** The general model is based on roles and allowed roles. Figure 2.3 illustrates the model. Each application has a set of associated *roles*. These roles represent groups of users with the same access rights. When a component is deployed, the administrator maps effective users from the local environment to roles declared by the component. This user-role mapping is based upon the credentials of the calling principal. For example, all users that are in an ‘admin’ group of the enterprise environment can be mapped to a role ‘administrator’ that is defined by a component. Roles introduce an environment-independent grouping of users to allow for greater scalability.

Second, an operation-allowed roles mapping is defined that tells which roles may execute which operations on the components. For example, the ‘administrator’ role can be mapped on all methods of a component, while the ‘customer’ role can be mapped on only one of those methods. The mapping is specified by grouping target operations that have the same set of allowed roles and stating per group what these allowed roles are.

The access control model combines these two mappings to determine whether the current principal has access. Whenever an operation is executed on a component, the component’s container checks whether the the current principal

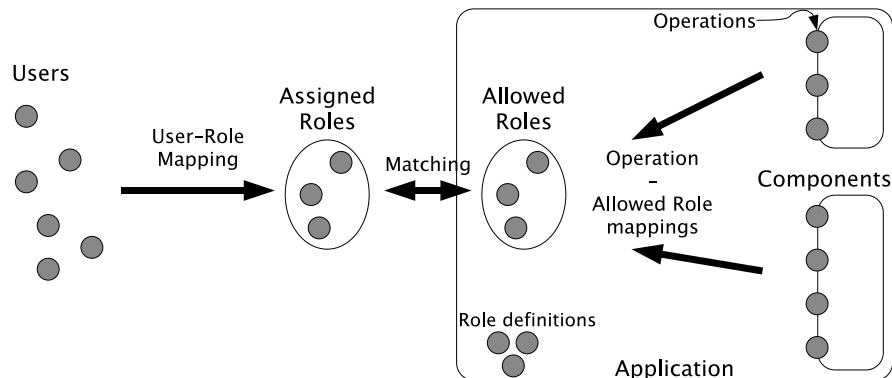


Figure 2.3: An overview of the J2EE role-based access control mechanism.

belongs to at least one of the allowed roles for the target operation. If this test succeeds, access is permitted; otherwise, access is denied.

When the policy for the operation-role mapping is too complex to be specified declaratively<sup>3</sup>, the access control can be enforced in the code of the component itself. Containers provide an API for retrieving the current caller's principal and for testing whether this principal is in a given role.

**EJB container** The operations that can be secured are the methods of an EJB<sup>4</sup>. Below, an example is shown of the part of the deployment descriptor that configures access control. The permissions are defined by `method-permission` elements of the deployment descriptor. The method-permission elements declare a number of roles and a number of methods. The semantics are that per method-permission all the specified roles are allowed to access all the specified methods. All methods must have been assigned to at least one method-permission. Two additional features are available: it is possible to disable access control for a method (and thus allowing everyone to access it) by stating that the method is `unchecked`, and it is possible to deny access to a method for everyone by adding the method to an `exclude-list`. Furthermore, a wildcard can be used to select all methods. It is impossible to impose access control on more course-grained or fine-grained targets than the bean's methods.

In the example below, customers may order products, internal customers and managers may order products and view sales numbers and the administrator may call any method. Furthermore, everybody may view products and nobody may call the bean's secret method.

```

...
<method-permission>
  <role-name>customer</role-name>
  <method>
    <ejb-name>eShopBean</ejb-name>
    <method-name>orderProduct</method-name>

```

<sup>3</sup>This typically arises when application-level information has to be used.

<sup>4</sup>On class level, not on instance level.

```

    </method>
<\method-permission>
<method-permission>
  <role-name>manager</role-name>
  <role-name>internalcustomer</role-name>
  <method>
    <ejb-name>eShopBean</ejb-name>
    <method-name>orderProduct</method-name>
  </method>
  <method>
    <ejb-name>eShopBean</ejb-name>
    <method-name>showSales</method-name>
  </method>
<\method-permission>
<method-permission>
  <role-name>admin</role-name>
  <method>
    <ejb-name>eShopBean</ejb-name>
    <method-name>*</method-name>
  </method>
<\method-permission>
<method-permission>
  <unchecked></unchecked>
  <method>
    <ejb-name>eShopBean</ejb-name>
    <method-name>showProducts</method-name>
  </method>
<\method-permission>
<exclude-list>
  <method>
    <ejb-name>eShopBean</ejb-name>
    <method-name>secretMethod</method-name>
  </method>
<\exclude-list>
...

```

**Web container** For web components, a similar mechanism is provided: the operations correspond to the HTTP requests (such as GET, POST, ...) on a servlet of a web application. In the deployment descriptor, a number of security constraints can be specified. Below, an example of a (part of a) deployment descriptor is given. Each security constraint consists of a number of web resources (**web-resource-collection**) and an authorization constraint (**authorization-constraint**). The web resources can specify the target to be secured using a number of URL patterns and/or by a number of HTTP methods. The authorization constraint consists of a number of roles. The semantics are that the resources of the resource collection can only be accessed by a principal that has at least one of the given roles. The default policy is to accept everything. The combination of multiple authorization constraints is done by taking the union of the roles that are permitted to use the resource.

The example allows only administrators to call GET and POST methods on pages in the /eshop/administration url.

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/eshop/administration/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>
```

## 2.2.4 Message Security

Security mechanisms are needed to protect application information as it travels along the network. Once outside the perimeter of the application, access control can no longer be applied because no software layer is available to access the information in network packets. As mentioned above, J2EE containers provide support for the SSL/TLS protocol to protect communications for confidentiality and integrity. The SSL/TLS protocols can be used for securing IIOP messages that are exchanged between EJB's and other J2EE components or for HTTP communication from a web client with a web container.

For EJB, the configuration of message security is done in the container. EJB containers use the IIOP protocol (of CORBA) and the IIOP messages can be protected using the CSIv2 [7] protocol, which uses SSL. All other J2EE containers (the web container and application client container) also support IIOP and IIOP over SSL. The container administrator has the choice of eight predefined public-key ciphersuites. Other vendor-specific SSL configurations can be provided as well.

Web components can specify a `user-data-constraint` in a security constraint of their deployment descriptor. In this constraint, web components can request that the web container should secure all communication with clients for the web resources that are associated with the security constraint. Web components can request integrity protection or both integrity and confidentiality protection for HTTP traffic. These constraints are enforced with the HTTPS protocol. The container administrator configures the HTTPS protocol for the container.

## 2.2.5 Audit

The J2EE specification doesn't support a secure auditing mechanism. The EJB specification does mention that containers may implement an audit trail that records the exception trace that is generated by each security exception.

## 2.3 CORBA

### 2.3.1 Introduction

In essence, the Common Object Request Broker Architecture (CORBA)[7] is a remote method invocation service that is used to simplify the programming of distributed (object-oriented) systems. CORBA allows objects that are programmed in different programming languages, deployed on different platforms and distributed in different networks to communicate. The *object request broker (ORB)* is the core middleware component that provides the basic remote invocation functionality. Each machine that hosts communicating objects runs an ORB. When a hosted object invokes an operation on a remote object, it sends its request to a client proxy object. This proxy sends the request to the ORB which takes care of all communication and wiring issues and forwards the request to the ORB at the target side. There, the invocation is reified and forwarded to a server-side proxy object (a skeleton), which finally calls the real operation on the target object. Every ORB can be extended by a number of *CORBA Services* that may be needed by distributed objects. Examples of CORBA services are the Naming Service, the Transaction Service and – to our interest – the Security Service. Figure 2.4 shows the CORBA architecture and the location of the Security Service.

The CORBA Security Service (CORBASec) standard [4] specifies the security model that is used to secure CORBA objects.<sup>5</sup> CORBASec makes a distinction between applications that are *security-unaware* and applications that are *security-aware*. Security-unaware applications (also called level 1) are applications that do not implement any security specific code. The deployment of such an application on a secured ORB suffices to implement security. The security configuration is performed in a declarative way by the administrator that deploys the CORBA object or component. Security-aware applications (also called level 2) implement the security functionality at least partially by themselves. The security logic is woven into the business logic of the application. The benefit of level 2 security is that the security implementation can be customized to support complex policies.

CORBASec uses the following general terminology:

**Principal** A human user or system entity that has the ability to use the resources of a system.

**Security Attribute** A piece of information that is associated with a principal and is used to represent security-related information. Examples of security attributes are *identity attributes* that represent the identity of the principal and *privilege attributes* that hold the access rights of the principal.

**Credentials** The credentials of a principal hold the security attributes.

**Policy Domain** A (security) policy domain defines a scope over which some security policy is imposed. The security policy of a policy domain can

---

<sup>5</sup>In the future, the standard will be replaced by the new Security Service Protocol (SECP), which is currently being adopted.

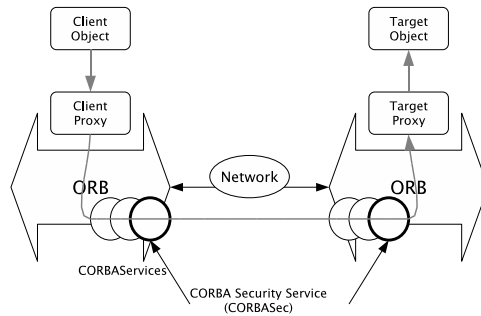


Figure 2.4: The CORBA architecture and the CORBA Security Service.

contain of many policies: eg for access control, secure invocation, authentication, delegation, auditing and non-repudiation. To enhance scalability, security policy domains can be divided in hierarchies or federations and they may overlap. The membership of objects to a policy domain is handled by the domain manager for that policy domain.

CORBA Sec covers a large number of security requirements [4, 7, 3, 8] that will be discussed in the following paragraphs. First, the common security features such as authentication, access control and message security will be discussed. Then, the more advanced security features non-repudiation and audit are handled.

### 2.3.2 Authentication

The CORBA Security Service specification doesn't state which exact authentication mechanisms can be used, but rather establishes a generic infrastructure for authentication that can be used for any possible authentication mechanism. The goal of authentication in CORBA Sec is to obtain a set of security attributes (contained in credentials). Some security attributes are available for every principal, while others can only be acquired by authentication. Examples of security attributes that typically require authentication are identity attributes and privilege attributes. Identity attributes represent the identities of a principal. Each principal can have a number of different identities that can be used by other security mechanisms such as access control or auditing. Privilege attributes are used for access control and represent the right to invoke a set of operations on a target object. In case of successful authentication, a **Credentials** object is created for the principal and registered with the **Current** object, which represents the execution context.

**Principal Authentication** The credentials of a principal must be established before CORBA invocations can be made. Some default credentials are established outside the CORBA infrastructure when the user logs on to the system. These credentials can be used transparently for authenticating the user. For improved flexibility, authentication can also take place within the CORBA infrastructure by level 2 applications.<sup>6</sup> For level 2 security, CORBA Sec specifies

<sup>6</sup>Either in the application itself or by a separate program.



Figure 2.5: A secure association and its policies.

an interface for authenticating a user to the ORB. A user typically authenticates via a *user sponsor*, in the simplest case a login application that asks the user for its username and password. This functionality has to be ‘glued’ to the initiating application by invoking it in the code. A principal that hasn’t established any credentials yet but has supplied its authentication data, can be authenticated by invoking the `authenticate` method on the `Principal Authenticator` object. This method expects a security name, specific authentication data and a number of requested privilege attributes. The requested privilege attributes serve for later usage in access control decisions. For a software entity the authentication data can be a key or a certificate, while for a human user this will typically be a password.

**Client-Server Authentication** A client that wants to invoke a method on a remote object securely, has to establish a *secure association*. This is a shared piece of information that represents the trust establishment between a client and target object. *Secure invocation policies* on the client and target side configure a secure association (See Figure 2.5). Each such policy can contain a number of *options* that specify which requirements the association has to implement. One of these options is the request for authentication. The client policy defines what is required from the target side and the target policy specifies the requirements from a client that invokes an operation on the target. It is also possible to specify an additional ‘supported’ policy that lists the functionality that the server supports but not requires.

The listing below shows an example of a (target) secure association policy in the commercial Orbix 6.2 ORB by Iona[38]. This fragment configures a server so that clients need to authenticate themselves in order to establish a secure association with it. It also sets a supported policy that can be applied when clients require integrity as well. Clients that do not support authentication cannot establish a secure association with the server.

```

policies:iiop_tls:target_secure_invocation_policy:requires =
    ["EstablishTrustInClient"];
policies:iiop_tls:target_secure_invocation_policy:supports =
    ["Integrity", "EstablishTrustInClient"];
  
```

**Delegation** When a chain of calls is made, an intermediate object can act on behalf of the calling principal, but only with respect to access control. This is achieved by *delegation* of a number of privilege attributes. CORBAsSec allows the enforcement of finegrained delegation policies. An initiator can specify *what* privilege attributes can be delegated, *where* these privilege attributes can be

used and *when* they can be used. *How* these privilege attributes can be delegated can be summarized as follows:

**No delegation** The intermediates may not use the initiators privilege attributes for further invocations, only for access control decisions upon reception.

**Simple delegation** The intermediate can use the credentials of the initiator in further invocations.

**Composite delegation** The intermediate may use the credentials of the initiator, but must supply its own credentials as well.

**Combined privilege delegation** The intermediate combines the initiator's privilege attributes with its own privilege attributes and supplies new credentials in further calls that contain the combined set of privilege attributes.

**Traced delegation** Delegation of the initiators credentials is allowed, but each intermediate has to append its own credentials when it makes the next call. This way, a trace of all intermediates is provided to the target.

### 2.3.3 Access Control

CORBASec features a very scalable access control model. The enforcement of all access decisions takes place in the ORB. Whenever an invocation is made, the client-side ORB as well as the target-side ORB run the code that enforces access control. An ORB's security service contacts the correct access policy that contains the access control rules. The central concepts that form the basis of the the access control model are: users, operations and target objects. The intent of access control is to enforce an authorization policy by restricting the operations a user can invoke on a target object. Scalability is achieved by providing a generic infrastructure that allows different grouping concepts to be implemented. Users are grouped using *privilege attributes*, operations are grouped using *rights* and targets are grouped using (*security domains*). All policies are expressed in terms of privilege attributes, rights and domains. By doing so, all environment-specific information is kept out of the policy.

**Privilege attributes** Each user has a number of privilege attributes. These attributes are part of the credentials a user gets when he or she is authenticated. Whenever a user invokes a target, its privilege attributes travel along with the request.<sup>7</sup>). What privilege attributes a user gets and how he/she gets them, depends on the CORBA implementation. A privilege attribute consists of a type and a value. CORBASec defines a number of default privilege attribute types such as 'AccessID' or 'Group', but new privilege attributes can be defined. By using privilege attributes, custom groupings can be made upon the set of users. For example, it is possible to put users in groups, roles, clearance levels or capability levels. Privilege attributes introduce a level of indirection on the set of users. Without privilege attributes, an access control policy would have to contain rules for each and every user.

---

<sup>7</sup>According to the delegation policy (see 2.3.2)

**Domains** Domains (more specifically, security policy domains) are used to impose a grouping of target objects. An object can belong to multiple domains. Domains can also form hierarchies. When a remote object reference is created by a client, the construction policy of the orb determines the policy domain of the created instance.

Domains introduce a level of indirection on the set of objects. Without domains, an access control policy would have to contain rules for every object.

**Rights** Rights are the central concept in the access control model. A number of default rights such as s (set), g (get), m (manage) and u (use) are defined, but it is possible to define new custom rights. On one hand, each operation of an interface has to specify a number of *required rights* and a combinator that can be *any* or *all*. On the other hand, each privilege attribute gets a number of *granted rights*. This last mapping is the actual access control policy. Since the scope of a security policy is a domain, one access control policy has to be defined per domain. Rights introduce a level of indirection on the set of methods. Without rights, an administrator would have to introduce rules for each method of an interface.

**Bringing it together** When we look at the whole picture, four mappings have to be defined:

- A mapping of user id to a set of privilege attributes
- A mapping of objects to a set of domains
- A mapping of operations to a set of required rights and possibly a combinator
- Per domain, a mapping of a privilege attribute value to a set of granted rights

The access control system works as follows: when a user authenticates to a system, he/she gets a number of privilege attributes. When an invocation on an object is made, the security service will look for the access control policy for the domain of the target object. In that policy, the service will lookup the granted rights the principal gets for all its privilege attributes. Finally, the service will match the granted rights with the required rights of the operation that is invoked, using the combinator.

For example, suppose we want to secure the following interface:

```
interface MyInterface
{
    void a();
    void b();
    void c();
};
```

Say we have one user called 'john'. When john is authenticated, he receives a privilege attribute of type 'Role' and with value 'employee'. Suppose we host the object on an ORB on the server 'myorb.mycompany.com'. We could let the policy domain correspond with the server hosting the ORB and the

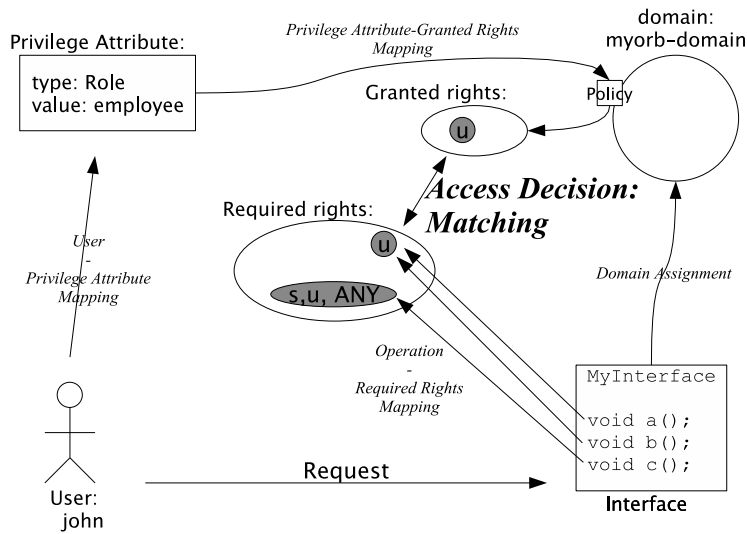


Figure 2.6: An example of how access control decisions are made in CORBASec.

object. We'll call this policy domain the 'myorb-domain'. The next mapping we need is an operation-required rights mapping. Let's say we have two rights: 'u' (from 'use') and 's' (from 'set'). We map the operations `a()` and `b()` to the required right 'u' and `c()` to the required rights 'u' and 's' with the ALL combinator. Finally, we need to configure one attribute-granted rights mapping for our 'myorb-domain'. Suppose this mapping gives principals with privilege attributes of the type 'Role' and value 'employee' the granted right 'u'. This configuration of the access control mechanism is illustrated in figure 2.6. When john makes a call in the 'myorb-domain', he'll get the granted right 'u'. This means that john can call methods `a()` and `b()` but not `c()`, because he does not have the required 's' right.

### 2.3.4 Message Security

Messages that are on the network and out of the safe control of the ORB can be protected for integrity and confidentiality. CORBASec uses the Common Secure Interoperability (CSIv2) protocol [7] for securing invocations on the wire.<sup>8</sup> The client and the target need to establish a secure association (see section 2.3.2) before they can make secure invocations. A secure association holds state information about the strength of protection for a particular invocation, depending on the policies of client and target. The protection strength of a secure association can be configured in the same policies as the client-server authentication options: the client and target secure invocation policies. The CSIv2 protocol is meant to be used with a transport-layer security protocol such as the SSL/TLS or SECIOP [4] protocols.

<sup>8</sup>This protocol is also used by J2EE.

### 2.3.5 Non repudiation

CORBA Security provides support for non-repudiation to security-aware applications (level 2). That is, applications need to invoke the non-repudiation service themselves. Not every CORBASec implementation has to implement the Non repudiation service.<sup>9</sup> Operations for generating and verifying evidence are specified in the interface `NRCredentials` (which inherits from the `Credentials` interface mentioned in section 2.3.2). A number of parties such as the delivery authority, adjudicator or evidence storage authority are included in a non-repudiation protocol, but are not explicitly supported CORBASec because they are mechanism-specific. The behavior of the non-repudiation service is specified in the non-repudiation policy. This policy specifies rules for the generation of evidence, for the verification of evidence and for adjudication.

**Non-repudiation tokens** All non-repudiation evidence is represented by a non-repudiation token. The contents of this token are mechanism specific, but a token always has a type (of action or event) and a time reference. A token might contain other mechanism-specific data such as a certificate. The non-repudiation service can be used to create tokens for non-repudiation of creation or receipt [11], but other types of evidence are also possible.

**Complete evidence** The non-repudiation service supports the notion *complete evidence*. This is evidence for which all data needed to verify it is incorporated in the evidence token. This way, everybody that has an evidence token can verify the token at all times. The non-repudiation policy may specify time constraints on when complete evidence can be formed. For example a time interval in which incomplete evidence may be completed.

### 2.3.6 Audit

It is possible to audit specific events to aid in the detection of security violations. A request can be audited at the client side and at the target side. The audit policies of the client and target define which events are audited. Two types of events can be audited: system events and application events. The format of audit records is unspecified.

**System events** System events are events that happen in the ORB or in a CORBA service such as the Security Service. Examples of system events are:

- Principal authentication
- Privilege changes
- Success or failure of object invocation
- Administration of security policies

---

<sup>9</sup>In fact, to our knowledge no implementations exist of the non-repudiation service so far.

**Application events** It is possible to declare application specific events that have to be audited. For example, a medical application may audit the modification of medical records of patients. Application specific events have an event family and a type within that family. Applications can define their own families and types.

**Audit selectors** To decide when an audit record needs to be written for an event, audit selectors are introduced. An audit selector is a trigger for auditing some request and can be based on information such as the interface implemented by the target, the value of an object reference, an operation name, client credentials, or the time or date of the request.

### 2.3.7 CORBA Component Model

**About the CORBA Component Model** The CORBA Component Model (CCM) [3] defines a component model that builds on CORBA and extends Sun's EJB component model. That is, it uses the same component-container architecture as EJB, and has equivalent operations for most EJB operations, but adds some features to components. The main difference with EJB is that CORBA components have *ports*. A port can be a

- *facet*: This is a provided interface the component exposes to its clients.
- *receptacle*: This is a required interface the component needs.
- *event source*: This is a producer of events.
- *event sinks*: This is a consumer of events.

Facets are linked to receptacles of other components and event sources and sinks are linked to an event channel. An EJB can be seen as a degenerate CORBA component with a fixed number of facets<sup>10</sup>. CORBA components run within a container that provides general services such as transactions, persistence and security. Under the hood, a CCM container uses an ORB for providing these services and making invocations on other components.

**Security in CCM** CCM containers use the CORBA Security Service for implementing security features. The CCM specification features an access control model that uses CORBASec rights for restricting access to components. The deployer of a component can declaratively assign required rights to individual operations of each facet in the deployment descriptor of the component. The container will enforce these access control rules transparently for the component using the CORBA Security Service of the underlying ORB. A container also provides an API that is similar to the security API of an EJB containers for retrieving the credentials of the current principal (`get_caller_principal()`) and for testing whether the current principal is in a given role (`is_caller_in_role()`).<sup>11</sup>

<sup>10</sup>The bean's remote, local, home and local home interfaces.

<sup>11</sup>This method explicitly mentions *role* as a security attribute although CORBASec and its access control model both make abstraction from specific security attributes.

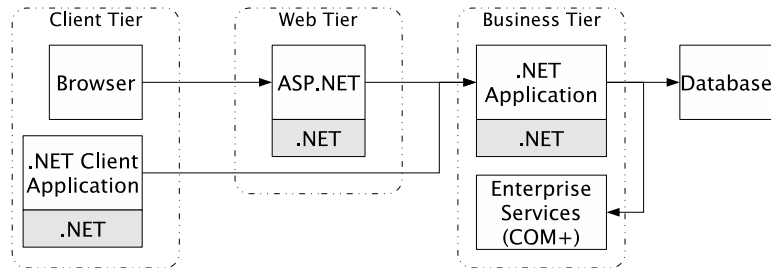


Figure 2.7: The architecture of a multitiered .NET application.

## 2.4 Microsoft .NET Framework

### 2.4.1 Introduction

The .NET Framework [18] is a development platform that is part of the Microsoft .NET initiative and consists of a language runtime (Common Language Runtime) and a set of frameworks and tools. The CLR runs applications compiled to bytecode and is comparable to the Java Virtual Machine with the major difference that multiple languages are supported. The .NET Framework offers a set of class libraries and frameworks that allow the development of a range of applications varying from plain client-side applications to multi-tiered web-based enterprise applications.

The following sections discuss the security features of the .NET Framework with a focus on the latter kind. Throughout the .NET Framework the following common terms are used:

**Identity** An identity represents the user on whose behalf a thread is running.

**Principal** A Principal represents the abstraction of a user and the roles to which a user belongs.

The web tier uses the ASP.NET technology while the business tier uses the .NET class libraries and the Enterprise Services technology (See figure 2.7) . The ASP.NET technology is strongly coupled to the Internet Information Services (IIS) webserver it runs on and the Enterprise Services technology provides a bridge between the .NET Framework and the COM+ application server. The security requirements that are handled by the .NET Framework are authentication, access control, message security and auditing[17, 13, 9].

### 2.4.2 Authentication

In the .NET Framework, users can be authenticated at the webtier with ASP.NET authentication and at the business tier with Enterprise Services authentication.

**ASP.NET Authentication** Authentication is implemented by different authentication modules. Three authentication modules are available:<sup>12</sup>

**Forms Authentication** Forms authentication allows a web application to show a custom HTML form for authenticating the user. When a user requests a page, the ASP.NET server will present the user an HTML form for entering its credentials instead. The user's credentials are then transmitted back to the application, where they must be authenticated. The specific authentication mechanism and storage of the user's credentials are undefined. The developer has to program the effective authentication requests in the application. Thus forms authentication is not really an authentication mechanism, but rather a user interface that gathers user's credentials.

**Windows Authentication** Windows authentication transparently authenticates the user by relying on the IIS webserver. This authentication mechanism is forwards the authentication responsibility to the IIS webserver which supports user authentication by one of the following techniques: basic HTTP authentication, digest HTTP authentication or windows authentication.

**Passport Authentication** This mechanism uses Microsoft's commercial .NET Passport authentication service [19]. This is a centralized web service for user authentication supporting single sign on.

**Enterprise Services Authentication** .NET applications can use enterprise services such as transaction management and security that are delivered by a COM+ application server. The infrastructure for integrating COM+ in the .NET Framework is called *Enterprise Services*. A .NET class can use enterprise services by subclassing the `ServiceComponent` class. The communication with the COM+ services uses RPC<sup>13</sup>. Clients of enterprise applications can be authenticated by RPC authentication mechanisms, which use the underlying Windows authentication systems (NTLM or Kerberos). The authentication can be done at the moment the connection is made, or with every RPC call.

**Impersonation and Delegation** ASP.NET supports a number of delegation options. *Impersonation* allows the local thread that serves a client to take on a completely different identity. The identity that will be used can be the identity passed by IIS or a fixed configured identity. Delegation allows the transfer of an identity across a network and is applied when ASP.NET uses impersonation and the hosts it calls are also configured for impersonation.

In COM+, a server can use the identity of a client when making further calls to other components. This is also called impersonation in COM+. Different levels of impersonation are defined:

**Anonymous** The client must remain anonymous to the server. No credentials may be transferred.

**Identify** The server may use the identity of the client only for access control checks.

---

<sup>12</sup>These mechanisms are hardcoded and cannot be extended.

<sup>13</sup>Because RPC is used by DCOM, the distribution model for COM+ applications.

**Impersonate** The server may use the identity of the client but with restrictions: if the server is on another host than the client, the server may only use the client's identity to components hosted on the same host as the server.

**Delegate** The server may use the identity of the client without restrictions.

The configuration of impersonation has to be negotiated. The client can give the level of impersonation it is willing to allow the server to use. The server can then present the identity of the client to other servers instead of its own according to the impersonation level the client has allowed. To do so, the method `CoImpersonateClient` has to be called.

### 2.4.3 Access Control

The .NET Framework defines two access control models: role-based access control for authorization based upon user identities, and evidence-based access control (or 'code access security') for authorization based upon code identity. These two access control models are completely separated.

**Role-based access control** Role-base access control in .NET is based on the *identity* and *principal* concepts. Every thread has a principal, which refers to one identity and knows the roles the identity belongs to. These abstractions can be used to implement application-level access control models. At run-time, an application can query a thread for its identities and roles by the `Principal.Identity` property and the `Principal.IsInRole(String role)` methods.

Instead of hardcoding the security-related calls, it is also possible to declaratively specify security constraints as custom attributes. These are annotations in the source code that are compiled into the assembly. Below is an example of this declarative style. The first method can only be accessed by the principal with the name 'john' on the machine 'myhost'. The second method can only be accessed by a principal that is in the role 'Administrator'.

```
class MyClass {
    [PrincipalPermissionAttribute(SecurityAction.Demand,
                                Name=@"myhost\john")]
    public void Method1() {
        //...
    }
    //implementation
    //...
}

    [PrincipalPermissionAttribute(SecurityAction.Demand,
                                Role="Administrator")]
    public void Method2() {
        //...
    }
    //implementation
    //...
}
}
```

**Evidence-based access control** Next to access control based on the user identity, .NET also implements an access control model that restricts actions based on the identity of the code that is executing. This model is primarily used to protect the host that is running downloaded code. Each assembly that is loaded by the CLR gets *evidence*. Evidence is information that is associated with a piece of code that serves to identify it. Examples of evidence are a url, a hash of an assembly or a certificate.

When an assembly is executed, it receives a number of *permissions*. A permission abstracts the authorization to do some protected action. It is possible to group permissions into sets and to intersect or unify permission sets. Once an assembly gets its permissions, these can only be limited further by additional constraints, but they cannot be extended by new permissions.

The .NET security system defines the mapping of evidence to permissions based upon a *security policy*. The security policy has four *policy levels*: enterprise, machine, user and application domain. Each policy level gives an assembly a number of permissions. The effective permission set an assembly gets, is the intersection of the permission sets of all different levels. Additional permission requests can be declared by using security custom attributes or by requesting them in the code.

Each policy level can be administered. A policy has three components: a tree of code groups, a list of named permissions and a list of policy assemblies. The code group tree implements the core decision function of the access control model. The permission sets that an assembly gets can be referenced by a chosen name. Thirdly, a list of policy assemblies is provided. These are assemblies that the policy evaluation function needs itself. By mentioning these assemblies explicitly, circular dependencies are avoided.

**ASP.NET authorization** Additional access control checks are available in ASP.NET through *authorization modules*. It is possible to restrict access on the aspx files that are being accessed by the webserver with the file authorization module. The URL authorization module can provide access checks based upon the URL of the web resource. The configuration of the URL authorization module uses a general configuration file that can allow or deny access to certain users or roles. The targets of this access control model are the contents of the directory the configuration file resides in. It is impossible to restrict access to individual web resources or HTTP methods. Below is an example of a configuration file that allows the user 'john' and the 'Administrator' role to access the web resources and denies access to the user 'mary' and all anonymous users:

```
<authorization>
  <allow users="john"/>
  <allow roles="Administrator"/>
  <deny users="mary"/>
  <deny users="?" />
</authorization>
```

**COM+ authorization** .NET applications that use Enterprise Services can make use of the COM+ role-based access control model. The roles in COM+ are completely independent of the .NET roles. The access control model restricts

access to resources to a number of specified roles. The granularity at which access can be restricted is the method-level. The configuration can be done either declaratively by using attributes or a configuration tool, or programmatically by a number of methods on the `ContextUtil` class such as `IsCallerInRole`.

#### 2.4.4 Message Security

Distributed applications cannot secure their communications transparently on the .NET platform. The .NET framework neither supports low-level SSL sockets nor Remoting over SSL. Only web applications that use IIS can use SSL out-of-the-box for message security. Other applications have to implement secure communications themselves using low-level cryptographic operations or rely on lower-layer message security with IPSec. .NET assemblies that use Enterprise Services can secure the messages they exchange with RPC encryption or integrity mechanisms.

#### 2.4.5 Audit

No real auditing services are defined by the .NET Framework, but security auditing can be simulated in a number of ways. The .NET Framework defines an API for event logging that can be used to programmatically write audit records to the event logging service of the underlying operating system (in practice, Windows 2000 or XP). Normal logging mechanisms on lower software layers can be also simulate security auditing. For example, IIS can log access to URL's and Windows can log access to certain files. However, these low-level mechanisms are not part of the .NET Framework.

## 2.5 Web Services Security

### 2.5.1 Introduction

According to Hartman [9], a web service is an “XML-based messaging interface to computing resources that is accessible via Internet standard protocols”. Normally, these messaging interfaces are defined by three XML-based standards: the Simple Object Access Protocol (SOAP) [39], the Web Services Description Language (WSDL) [40] and the Universal Description Discovery and Integration standard (UDDI) [32]. SOAP describes a wiring standard for remote object invocations and is comparable with Java RMI, CORBA or .NET Remoting to a certain extent, but uses XML to represent its messages and usually relies on a high-level internet protocol such as HTTP for delivering them. SOAP is the protocol that is used to make invocations on web services. UDDI is a directory for web services that can be used to publish, search and discover web services that implement a certain feature. WSDL is the language that defines the interface itself and can intuitively be seen as “an IDL for web services”.

Web services allow applications of different companies to interact with each other in a loosely coupled manner over the internet. In theory, each application can become a client of each other application. It is easy to see that security issues are of great importance for web service infrastructures. The domain of

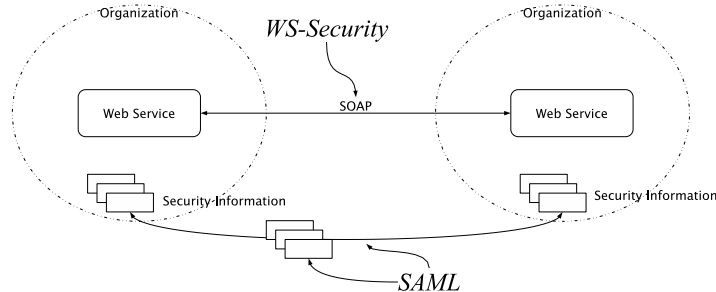


Figure 2.8: A web services infrastructure and the position of the WS-Security and SAML standards.

web services security is dominated by two standards from the Oasis consortium, which are described in this section:

- Web Services Security (WS-Security) [34]
- Security Assertion Markup Language (SAML) [33]

WS-Security describes a way to secure SOAP messages. SAML defines an infrastructure that allows interoperable interpretation of security-related information on different platforms.

## 2.5.2 WS-\* family of security standards

The most important set of security standards and specifications for web services has been elaborated by an alliance of a number of major software vendors mainly consisting of Microsoft and IBM. The main standard is WS-Security. Other standards build on WS-Security to provide additional security functionality such as the establishment of trust relations or invocation contexts.

**WS-Security** WS-Security is a standard for end-to-end SOAP-level message security that builds on the XML Signature [43] and XML Encryption [42] standards to provide integrity and confidentiality for SOAP messages or parts of SOAP messages (see Figure 2.8). The standard also provides a generic mechanism for attaching *security tokens* to the header of a SOAP message. These security tokens are defined as broad as possible and contain “statements that the client makes” (also called claims). A token can be a X.509 certificate or Kerberos ticket, as well as a privilege or role membership. Security tokens are meant to be extensible.

SOAP messages that are secured by WS-Security carry one or more `<Security>` elements in their header. Below is an example of a SOAP message that is secured for integrity (the example is taken from the WS-Security specification[34]). The body of the message shows that it is a StockSymbol message that has “QQQ” as value. The header of the message contains the security information that is appended by WS-Security. First, a binary security token is defined. In this case, this token is a X509 certificate. The token gets a name (“X509Token”)

so it can be referenced later on. After the security token definition, an XML Signature element is found. This element contains the digital signature of the message body using the certificate defined above.

```
<?xml version="1.0" encoding="utf-8"?>
<S11:Envelope xmlns:S11="..." xmlns:wssse="..."
  xmlns:wsu="..." xmlns:ds="..." >
  <S11:Header>
    <wssse:Security>
      <wssse:BinarySecurityToken ValueType="...#X509v3"
        EncodingType="...#Base64Binary"
        wsu:Id="X509Token">
        MIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
      </wssse:BinarySecurityToken>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
            "http://www.w3.org/2001/10/xml-exc-c14n#" />
          <ds:SignatureMethod Algorithm=
            "http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
          <ds:Reference URI="#myBody">
            <ds:Transforms>
              <ds:Transform Algorithm=
                "http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm=
              "http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>EULddytSo1... </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>
          BL8jdfToEb1l/vXcMZNNjPOV...
        </ds:SignatureValue>
        <ds:KeyInfo>
          <wssse:SecurityTokenReference>
            <wssse:Reference URI="#X509Token" />
          </wssse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wssse:Security>
  </S11:Header>
  <S11:Body wsu:Id="myBody">
    <tru:StockSymbol
      xmlns:tru="http://www.fabrikam123.com/payloads">
      QQQ
    </tru:StockSymbol>
  </S11:Body>
</S11:Envelope>
```

Why is a SOAP-level security standard needed instead of low-level security protocols such as SSL? The SSL standard only provides point-to-point security.

That is, a security context can only be established between two nodes that are directly connected to each other. However, interacting web services are not always directly connected (SOAP messages can for example be routed). Furthermore, SSL can only be used for in-transit security and it only provides coarse-grained all-or-nothing security. Consequently, SSL is not powerful enough for securing web services.

**Other standards from the WS-family** WS-Security is a building block for a number of other security specifications for web services. As of writing, much of these specifications are not yet standardized and only have early draft versions. Therefore we will only give a short overview:

**WS-Policy [37]** This specification defines a framework for expressing general purpose policies for web services.

**WS-Security Policy [28]** This specification extends WS-Policy to express security policies of web services.

**WS-Trust [31]** This specification specifies protocols and representations by which communicating parties can establish trust relations as a basis for secured SOAP messages.

**WS-Privacy [21]** defines a privacy model for web services.

**WS-SecureConversation [30]** allows interoperating web services to establish security contexts that hold keys for multi-message secure conversation.

**WS-Federation [29]** allows federation of security information and mechanisms between different security domains.

**WS-Authorization [21]** allows the management of authorization information and policies.

### 2.5.3 SAML

When web services of different companies interoperate, they probably use different platforms and by consequence different security services. The goal of the SAML standard is to define a framework for exchanging security context information between different (online) parties. The standard defines a way to represent this information, a set protocols for exchanging the information, and bindings for integrating the SAML protocols with existing technologies. SAML is not web services-centric by definition, but web services were one of the driving factors for creating the standard.

SAML limits its scope to the representation and exchange of information related to authentication, security attributes and authorization. The SAML standard defines a secure interoperability architecture based upon the exchange of these kinds of security-related information. On the one hand, there are *assertions*. An assertion is a document that holds security-related information and is assertable by a trusted third party. There are assertion types for authentication, attributes and authorization. For example, an authentication assertion can state that a certain principal has been authenticated, an attribute assertion

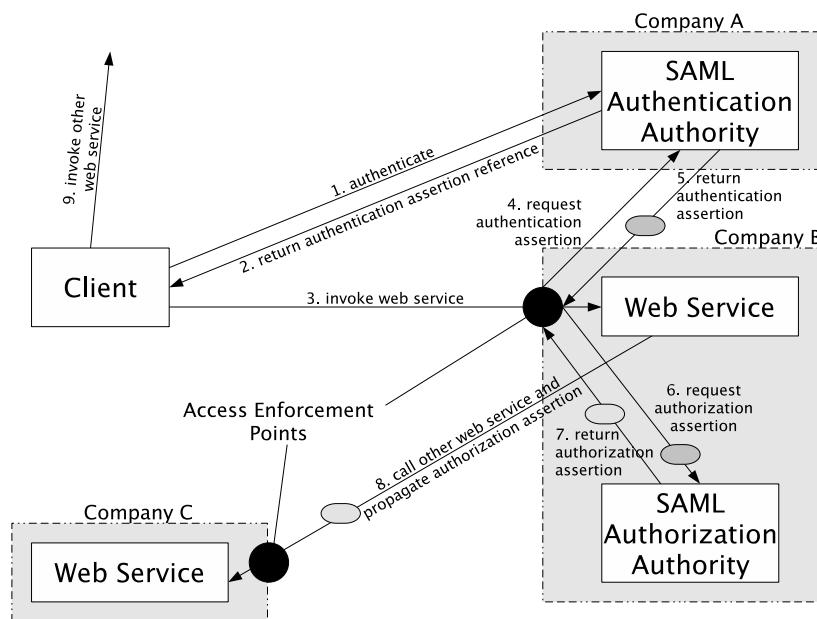


Figure 2.9: An example of a SAML interaction that uses authentication and authorization assertions.

can state that the principal in a certain role and an authorization assertion can state that a subject has the permission to access a resource. On the other hand, there are *SAML authorities*. Authorities are producers of assertions. There are authority types for each assertion type. SAML doesn't specify how the authorities must be implemented, but it does specify the protocol that is used to retrieve the assertions from the authorities. This protocol is defined on top of SOAP.

A typical use scenario of SAML is illustrated in Figure 2.9. In this example scenario, a client uses two web services. Company B's web service will in turn use company C's web service in the client's name. Company A provides SAML authentication services.

1. The client authenticates to company A and sends a SAML request for an authentication assertion.
2. After successful authentication, company A's SAML authentication service creates an authentication assertion and returns a reference to the client. The authentication assertion looks like this:

```
<saml:Assertion
  xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
  MajorVersion="1" MinorVersion="1"
  AssertionID="..."
  Issuer="https://saml.company-a.com/"
  IssueInstant="2005-02-05T17:05:37.795Z">
  <saml:Conditions
    NotBefore="2005-02-05T17:00:37.795Z"
```

```

    NotOnOrAfter="2005-02-05T17:10:37.795Z"/>
<saml:AuthenticationStatement
  AuthenticationMethod=
    "urn:oasis:names:tc:SAML:1.0:am:password"
  AuthenticationInstant=
    "2005-02-05T17:05:17.706Z">
<saml:Subject>
  <saml:NameIdentifier Format="urn:oasis:names:
    tc:SAML:1.1:nameid-format:emailAddress">
    client@clientscompany.org
  </saml:NameIdentifier>
  <saml:SubjectConfirmation>
    <saml:ConfirmationMethod>
      urn:oasis:names:tc:SAML:1.0:cm:artifact
    </saml:ConfirmationMethod>
  </saml:SubjectConfirmation>
</saml:Subject>
</saml:AuthenticationStatement>
</saml:Assertion>

```

As can be seen, it includes information about the issuer, a validity period, the authentication method and of course the subject identifier.

3. The client now makes an invocation on the web service and passes the reference to authentication assertion it just received.
4. Company B wants to enforce access control on its web service and has its own SAML-compatible authorization authority. When the client invocation arrives at Company B, the access enforcement point of Company B makes a SAML request for retrieving the authentication assertion for the client and sends it to Company A's SAML (trusted) authentication service.
5. Company A returns the authentication assertion for the client to Company B's enforcement point.
6. Company B's enforcement point now constructs an SAML request for an authorization attribute for the client. The authentication assertion is included in the request and the request is sent to Company B's SAML authorization authority.
7. The authorization authority verifies the authentication assertion for the client and makes an access decision. It constructs an authorization assertion that contains its decision and returns it to the enforcement point. The authorization assertion looks like this:

```

<saml:Assertion
  xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
  MajorVersion="1" MinorVersion="1"
  Issuer="https://saml.company-b.com" ... >
<saml:Conditions .../>
<saml:AuthorizationDecisionStatement

```

```

    Decision="Permit"
    Resource="https://company-b.org/theweb-service">
  <saml:Action>...</saml:Action>
  <saml:Subject>...</saml:Subject>
</saml:AuthorizationDecisionStatement>
</saml:Assertion>

```

8. The enforcement point sees in the received assertion that the client may use the service and allows access to the client.

Now, the web service itself wants to invoke another web service in Company C in the name of the client. It makes the call and passes the authorization assertion for the client. Company C's access enforcement point receives the invocation and sees the authorization assertion. It trusts Company B and knows that it can allow access for calls that were allowed to access Company B's web service.

9. A couple of minutes later, the client wants to use a web service from a completely different company that also trust Company A's authentication services. It makes a request and appends the original reference to the authentication assertion. Now, the receiving web service can contact Company A's authentication authority to get the assertion and it can assume that the client is authenticated.

We conclude with some final remarks about SAML. This scenario is just one possible application of SAML. The SAML specification does not introduce an obligation to use this specific architecture, it rather provides a framework for building these kinds of secure interoperability architectures. For example, NASA's Cardea system [15] is a SAML-based distributed access control system for grid computing. However, SAML does specify two *profiles* for supporting single sign on with normal web browsers on the client side: the Browser/Artifact profile and the Browser/POST profile. In fact, single sign on is one of the main applications of SAML. The scenario has demonstrated this: the client only authenticated to Company A and can access two different and independent web services without having to reauthenticate.

#### 2.5.4 Other web service-related security standards

Besides WS-Security and SAML, some other security standards related to web services have been established. The Extensible Access Control Markup Language (XACML) [35] is an XML-based access control language. It is meant to be integrated with SAML to allow more expressive access control requests, but its scope is wider than SAML alone. The XML Key Management Specification (XKMS) [41] defines a web service interface for public key infrastructures. More precisely, it allows the distribution and registration of public keys by means of a SOAP-based protocol.

## Chapter 3

# Security in Middleware for Mobile Devices

### 3.1 Introduction

Mobile and embedded devices are becoming more powerful. Increased processing power and communication facilities have given them the potential to run elements of large-scale distributed applications. Various middleware platforms have been developed that try to port the client-side of existing platforms to this family of devices. As a consequence, security on these embedded devices becomes an important issue.

On the other hand, these devices are still limited when compared to normal hosts and these limitations have an impact on the support for security on the middleware platforms that they support. Therefore, the security functionality of these platforms mainly concentrates on device protection. This section explores the security mechanisms of the most important middleware platforms for embedded devices. First, the security features of the operating systems themselves are considered. Then, security support in the three major embedded middleware platforms J2ME, the .NET Compact Framework and the mobile versions of CORBA and CCM are discussed.

### 3.2 Operating system

In normal desktop and server operating systems there are always enough resources to implement the security primitives that are used by the security services of the the middleware layer. On embedded devices that have to run parts of a distributed application, this guarantee cannot be made by the operating systems. Because of the restricted resources these devices have, common security functionality such as encryption is limited. Furthermore, the security functionality of these operating systems also affects the security of a distributed application that is partly deployed on an embedded device.

This section gives an overview of the security mechanisms that are offered by common operating systems for mobile devices, with a focus on Palm OS [10] and Windows CE .NET [20].

### 3.2.1 Device access

Operating systems for mobile devices often provide support for *locking* the device. This means putting the device in an unusable state as long as the user can't authenticate him/herself. Since these devices only have a single user, a password without username is used for authentication. The locking can take place each time the device is powered off, after a period of inactivity, or at a given time.

### 3.2.2 Cryptographic Libraries

Most mobile operating systems ship with cryptographic libraries that are built using a *service provider* architecture. Examples are Palm OS's Cryptography Provider Manager and Windows CE's CryptoAPI. They implement a collection of basic cryptographic algorithms for encryption, hashing, and signing and form the basis for most of the security services above the operating system layer.

Although these libraries are extensible, by default they usually come with a much more restricted set of cryptographic algorithms than their desktop counterparts and the supported key strengths are limited. For example, today neither Palm OS nor Windows CE support the AES algorithm.

### 3.2.3 Data Security

Some operating systems also offer an API for the secure storage of data on the mobile device. These services build on the cryptographic libraries of the operating system and provide a high-level abstraction for encrypting data that is written to the storage memory of the device.

### 3.2.4 Message security

Almost all mobile operating systems implement the SSL protocol for securing communications on the transport layer and for server-side or mutual authentication. This is important since it is the only component that is needed on the client side for interaction with a centralized enterprise application in a secure way.

## 3.3 J2ME

The Java 2, Mobile Edition is the Java edition for mobile and embedded devices. J2ME targets two kinds of embedded devices: very resource-limited devices such as phones, pagers or pda's and larger devices such as set-top boxes. Each of these two categories has their *configurations*. A configuration is a collection of a virtual machine and low-level APIs. J2ME defines two configurations: the connected limited device configuration (CLDC, [24]) and the connected device configuration (CDC, [22]). The former runs on a limited virtual machine, the latter runs on a normal virtual machine. Since within each configuration a diversity of devices is targeted, a configuration is complemented by a device-specific *profile*. The most important J2ME profile is the Mobile Information Device Profile (MIDP) [23] for CLDC. This is the profile that is used for small connected devices that are capable of displaying a GUI. Examples of such devices

are mobile phones or pda's. MIDP is implemented for a range of operating systems including Palm OS and Symbian. The security architecture of the CDC configuration is quite similar to normal J2SE security. Therefore, we will mainly discuss MIDP security in this section. The security architecture for MIDP strongly focuses on device protection. It is based on a sandboxing model that restricts access to resources based on code authentication. In a distributed environment, network traffic can also be secured. The following sections introduce the security features of MIDP.

### 3.3.1 Protection Domains

All applications that are downloaded and run on a MIDP-enabled device operate in a sandbox. Access to sensitive API's is restricted based upon the trust level that the device has in the application's source and integrity. Each application executes in a *protection domain*. Applications whose integrity and source cannot be verified run in the *untrusted domain*. These applications are allowed to call a well-defined but very limited set of API's that should not be able to harm the device. Applications that are trusted execute in their own protection domain that allows them to execute more dangerous API's (for example classes of the `javax.microedition.io` package).

A protection domain is defined by a set of *permissions*. Permissions in J2ME are more restrictive than in J2SE. A permission is an authorization to use some class and is represented by the package or full classname name of the class. Permissions can be either *user permissions* or *allowed permissions*. User permissions require user interaction before they can be used, while allowed permissions are invisible for the user. User permissions ask the user if the permission can be granted on a configurable moment: the first time it is requested, at the beginning of each session, or every time.

When the application is installed it is placed in a protection domain, depending on the trust level that the system has in the application. Applications can explicitly request the permissions they require in their metadata. When such an application is installed, the J2ME environment sees if the acquired protection domain of the application has sufficient permissions to meet the required permissions. If it has, the installation proceeds; otherwise the installation is aborted by a security exception. Permissions that are not explicitly requested are checked at run time.

### 3.3.2 Application Signing

Applications must gather a trust level to be allowed to a protection domain. MIDP defines a certificate-based way of establishing this trust. The application carries a digital signature of the archive file it is distributed in. The distributor of the application signs the jar file of the application and appends the signature and its certificate in the metadata of the application.

When the application is installed, the system checks the signature and can then decide to put it in a trusted or in the untrusted protection domain.

### 3.3.3 Communications Security

MIDP supports the SSL/TLS protocols for securing communications over a network connection. However, only HTTP traffic can be secured.

## 3.4 Microsoft .NET Compact Framework

The .NET Compact Framework [1] is a small version of the .NET Framework targeting mobile and embedded devices that lack the resources to run a full version of the framework. To adapt the framework to this kind of limited devices, only a subset of the class library of the full .NET Framework is included. Furthermore, the .NET Compact Framework has its own Common Language Runtime implementation that is optimized for efficiency and size constraints. As with the .NET Framework, the compact edition is only implemented for Microsoft's Windows CE operating system, although in theory applications written for it are portable to other operating systems.

Because of the constraints of these devices, several security features that are present in the full framework are not available in the compact framework. The security architecture and features of the .NET Compact Framework are introduced in the rest of this section. First authentication is discussed, then the code-based access control model is presented, and finally communications and data security are introduced.

### 3.4.1 Authentication

It is possible to communicate over a network using the HTTP protocol. When setting up an HTTP connection, it is possible that the webserver requests authentication. The compact framework provides an interface for passing authentication data using basic HTTP authentication or Windows Integrated authentication. This last authentication protocol is a proprietary microsoft protocol that is supported by the IIS webserver and makes use of Windows accounts. Basic authentication transports credentials in cleartext while Windows Integrated authentication encrypts them. No other forms of authentication are supported.

### 3.4.2 Access Control

Since the .NET Compact Framework contains an implementation of the CLR, the code access security architecture (see section 2.4.3) is also implemented. In the full version of the framework, this model allows administrators to impose selective access restrictions to executables based on a variety of code authentication mechanisms. In the .NET Compact Framework this model is implemented, but it is impossible to make use of it because no software layer is built around it. The framework contains one hard-coded code group (`ALL_CODE`) that gives all permissions to its members and all applications that are loaded are put in this code group.

The role-based access control model defined by the full .NET Framework, is not implemented in the compact editions because the framework typically runs on one-user devices. As a consequence, currently no form of local access control is applied.

### 3.4.3 Data and Communication Security

The .NET Compact Framework does not support the cryptographic libraries from the full version of the framework. If low-level cryptography is needed, developers have to use Windows CE's CryptoAPI or a third party cryptographic library.

Communications can be secured by the SSL/TLS protocols, but only for HTTP traffic. When setting up an HTTP connection to a `https://` url, SSL will be used automatically. Low level secured sockets are not programmable by the developer.

## 3.5 CORBA

The Object Management Group has defined two standards for implementing CORBA environments on embedded devices: minimumCORBA [5] is a specification for embedded ORBs, the Lightweight CORBA Component Model is a specification for an embedded version of the CORBA Component Model. Some embedded implementations do not follow these standards for implementing CORBA on embedded devices and customize a normal ORB (and the security service) for these devices. For example, ObjectSecurity has ported the MICO ORB and Security Service to a Windows CE-based pda [16]. In the rest of this section, we will focus on security considerations for minimumCORBA and Lightweight CCM.

### 3.5.1 minimumCORBA

The minimumCORBA specification defines a subset of the normal CORBA specification that can be used to build small footprint ORBs for embedded devices. Various aspects such as dynamic invocations and the interface repository are omitted in comparison with the full CORBA specification. A minimumCORBA ORB retains enough of CORBA to remain interoperable with other ORBs. Since minimumCORBA only defines an ORB, optional CORBA Services such as the Security Service remain implementable for minimumCORBA.

There are not much implementations of minimumCORBA; to our knowledge there is one ORB from the Bionic Buffalo Corporation [2] that implements level 1 security (for security-unaware applications) of the security service.

### 3.5.2 Lightweight CORBA Component Model

The Lightweight CORBA Component Model specification defines a subset of the full CCM. Lightweight CCM completely removes all security functionality from CCM.

# Bibliography

- [1] S. Wheelwright A. Wigley. *.NET Compact Framework: Core Reference*. Microsoft Press, 2003.
- [2] Bionic Buffalo Corporation. Tatanka Gibraltar Minimum CORBA Implementation. <http://www.tatanka.com/prod/info/gibraltar.html>.
- [3] Object Management Group. CORBA Components Specification, Version 3.0. <http://www.omg.org/cgi-bin/apps/doc?formal/02-06-65.pdf>, 2002.
- [4] Object Management Group. CORBA Security Service Specification, Version 1.8. <http://www.omg.org/cgi-bin/apps/doc?formal/02-03-11.pdf>, 2002.
- [5] Object Management Group. Minimum CORBA Specification, Version 1.0. <http://www.omg.org/cgi-bin/apps/doc?formal/02-08-01.pdf>, 2002.
- [6] Object Management Group. Java Language Mapping to OMG IDL Specification. <http://www.omg.org/cgi-bin/apps/doc?formal/03-09-04.pdf>, 2003.
- [7] Object Management Group. Common Object Request Broker Architecture specification, version 3.0.2. <http://www.omg.org/cgi-bin/apps/doc?formal/04-03-01.pdf>, 2004.
- [8] Hartman, Flinn, and Beznosov. *Enterprise Security with EJB and CORBA*. Wiley, 2001.
- [9] Hartman, Flinn, Beznosov, and Kawamoto. *Mastering Web Service Security*. Wiley, 2003.
- [10] PalmSource Inc. Palm OS. <http://www.palmsource.com/palmos/>.
- [11] ISO. Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture, 1989.
- [12] Pankaj Kumar. *J2EE Security For Servlets, EJBs and Web Services*. Prentice Hall, 2004.
- [13] Bert Lagaisse. Een Vergelijkende Studie van J2EE en .NET Architectuur en Beveiliging van Bedrijfstoeepassingen. Master’s thesis, KULeuven, 2003.

- [14] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User Authentication and Authorization in the Java Platform. In *Proceedings of the 15th Annual Computer Security Applications Conference*, 1999. <http://java.sun.com/products/jaas/>.
- [15] Rebekah Lepro. Cardea: Dynamic access control in distributed systems. Technical report, NASA Advanced Supercomputing (NAS) Division, 2003.
- [16] ObjectSecurity Ltd. MICO CORBA Implementation. <http://www.mico.org/>.
- [17] Microsoft. An Overview of Security in the .NET Framework. <http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dn%netsec/html/netframesecover.asp>.
- [18] Microsoft. .NET Framework Technology Overview. <http://msdn.microsoft.com/netframework/technologyinfo/overview/>.
- [19] Microsoft. .NET Passport Service. <http://www.passport.net>.
- [20] Microsoft. Windows CE .NET. <http://msdn.microsoft.com/embedded/prevver/ce.net/>.
- [21] Microsoft and IBM. Security in a Web Services World: A Proposed Architecture and Roadmap. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwsse%cur/html/securitywhitepaper.asp>, April 2002.
- [22] Sun Microsystems. Connected Device Configuration, Version 1.0a, 2002.
- [23] Sun Microsystems. Mobile Information Device Profile Specification, Version 2.0, 2002.
- [24] Sun Microsystems. Connected Limited Device Configuration, Version 1.1, 2003.
- [25] Sun Microsystems. Enterprise JavaBeans Specification, Version 2.1, 2003.
- [26] Sun Microsystems. Java 2 Platform Enterprise Edition Specification, Version 1.4, 2003.
- [27] Sun Microsystems. Java Servlet Specification, Version 2.4, 2003.
- [28] Anthony Nadalin et al. Web Services Security Policy Language Specification, Version 1.0 Draft. <http://www.ibm.com/developerworks/webservices/library/ws-secpol/index.h%tml>, December 2002.
- [29] Anthony Nadalin et al. Web Services Federation Language Specification, Version 1.0 Draft. <ftp://www6.software.ibm.com/software/developer/library/ws-fed.pdf>, July 2003.
- [30] Anthony Nadalin et al. Web Services Secure Conversation Language Specification, Version 1.1 Draft. <ftp://www6.software.ibm.com/software/developer/library/ws-secureconvers%ation.pdf>, May 2004.

- [31] Anthony Nadalin et al. Web Services Trust Language Specification, Version 1.1 Draft. <http://www-106.ibm.com/developerworks/library/specification/ws-trust/>, May 2004.
- [32] Oasis. Universal Description Discovery and Integration Specification, Version 2.0. <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>, 2002.
- [33] Oasis. Security Assertion Markup Language Specification, Version 1.1. <http://www.oasis-open.org/committees/download.php/3400/oasis-sstc-saml%-1.1-pdf-xsd.zip>, 2003.
- [34] Oasis. Web Services Security: SOAP Message Security, Version 1.0. [http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-se%curity-1.0.pdf](http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf), 2004.
- [35] Oasis. eXtensible Access Control Markup Language (XACML) Version 2.0. [http://docs.oasis-open.org/xacml/access\\_control-xacml-2\\_0-core-spec-cd-%04.pdf](http://docs.oasis-open.org/xacml/access_control-xacml-2_0-core-spec-cd-%04.pdf), December 2005.
- [36] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
- [37] Jeffrey Schlimmer et al. Web Services Policy Framework Specification, Draft Version. <ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf>, September 2004.
- [38] Iona Technologies. Orbix, Version 6.2. <http://www.iona.com/products/orbix.htm>, 2004.
- [39] W3C. Simple Object Access Protocol, Version 1.2 Recommendation. <http://www.w3.org/TR/soap/>.
- [40] W3C. Web Services Description Language, Version 1.1. <http://www.w3.org/TR/wsdl>, 2001.
- [41] W3C. XML Key Management Specification (XKMS), W3C Note 30 March 2001. <http://www.w3.org/TR/2001/NOTE-xkms-20010330/>, 2001.
- [42] W3C. XML Encryption Syntax and Processing. <http://www.w3.org/TR/xmlenc-core/>, 2002.
- [43] W3C. XML Signature Syntax and Processing. <http://www.w3.org/TR/xmlsig-core/>, 2002.