Reusable Components for Developing Security-Aware Applications

Stefan Probst, Wolfgang Essmayr, Edgar Weippl
Software Competence Center Hagenberg
{stefan.probst, wolfgang.essmayr, edgar.weippl}@scch.at

Abstract

Today, security is considered to be an important aspect of multi-tier application development. Thoroughly researched concepts for access control exist and have been proven in mainframe computing. However, they are often not used in today's development of multi-tier applications. One reason may be the lack of appropriate reusable components that support application developers that frequently have to re-invent the wheel when it comes to access controls. The goal of this paper is to promote awareness of security issues when developing applications and to illustrate a suitable approach for that. Our framework called GAMMA (Generic Authorization Mechanisms for Multi-Tier Applications) offers several authentication, access control, and auditing mechanisms. Access control models can be combined or used simultaneously in order to provide application-specific and highly customizable mechanisms. Moreover, due to its component-based structure, new security models and additional approaches for authentication or auditing can easily be added.

1. Introduction

Security achieves more and more acceptance as „enabling technology" within our Inter-networked society. Currently, we experience a lot of security infringements being increasingly accompanied with substantial financial losses of the attacked enterprises. Within modern, multi-tier application architectures the consistent and transparent enforcement of security mechanisms especially at the application- and business-logic levels can be a decisive measure against this trend (compare [1], [2]).

As illustrated in Figure 1 there are several levels of security mechanisms. Reusable components for the development of security-aware applications are available especially for the lower levels of IT security, namely, cryptography and communication security. At the higher levels, i.e. authorization models, access controls, authentication, and auditing, adequate components require a specific platform and/or architecture (e.g. .NET, J2EE) and are most of the time not expressive enough respectively cannot be sufficiently adapted to complex application requirements (see [3]). Consequently, the enforcement of high-level security mechanisms at the application- and business-logic layers results in a practice of permanently re-inventing the wheel.

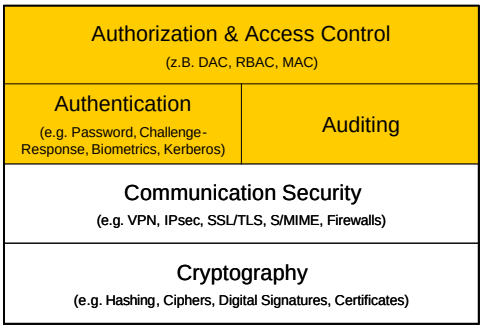

Figure 1 Levels of security mechanisms.

In this work, we present the design and Java implementation of a security framework called GAMMA (Generic Authorization Mechanisms for Multi-Tier Applications). The framework contains a set of components offering a range of high-level security mechanisms including discretionary access controls (DAC), role-based access controls (RBAC), the possibility to use multiple concurrent authorization models, support of negative authorization, or arbitrary constraints such as separation-of-duty, which are ready to be used for application development. The design especially focuses on platform and architecture independency, allowing us to transfer the implementation to other platforms and programming languages in future.

The remainder of this paper is structured as follows: section 2 discusses general issues for a generic security framework and provides related work. Section 3 presents the set of components designed for offering high-level security mechanisms. Section 4 discusses realization issues encountered during implementation of the

GAMMA framework. Furthermore, section 5 briefly sketches an example showing the application of the framework. Finally, section 6 concludes and gives an outlook on future activities.

2. Generic security framework

Modern software applications are realized using a multi-tier architecture. The software is divided into several tiers or layers according to its functionality. Each layer is able to communicate with the underlying or superior layer via a well-defined interface.

Figure 2 illustrates the layered architecture conceived for providing high-level security mechanisms in multi-tier environments.

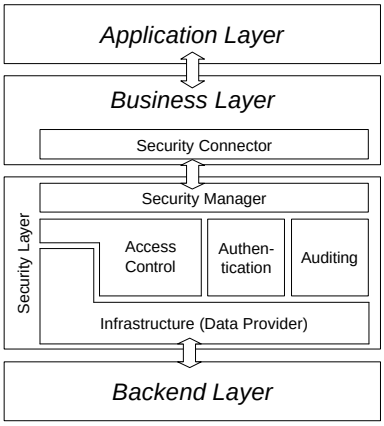

Figure 2 Architecture for multi-tier security.

In particular, a security layer is established in between the business and backend layers of the architecture. The security layer has a distinct interface to the backend layer (i.e. data providers) exchanging security data with backend systems. Furthermore, a particular component (i.e. a security connector) is offered, which is to be used as entry point from the business- (or application) to the security layer. Finally, the security layer contains high-level components for security coordination in general (i.e. the security manager), for the provided security mechanisms (i.e. authentication, access control, auditing), and for the infrastructure required to enforce the aforementioned security mechanisms. A detailed description of the components is given in section 3.

The security layer consists of a set of classes with a common focus (i.e. security) and can thus be called framework. We categorize the potential target users of the framework into five groups as illustrated in Figure 3.

The users of business applications benefit from using applications that contain high-level security mechanisms. The connection between the users and the framework is given by an increased confidence in the software they are using. However, there is no direct link between the users and the framework. Business application developers are interested in developing applications using the framework for integrating high-level security mechanisms in their business applications. If they input anything into the framework, it is in terms of new ideas and visions that will be realized by framework architects, who are responsible for the design, implementation, maintenance, and further development of the security framework itself. Model providers are in charge of introducing new access control models into the framework. This can be done without modifying the framework kernel, which will only be changed by framework architects. Security administrators are responsible for setting up the framework's mechanisms. Primarily, their work consists of realizing the security policy in defining the access rights and the mapping between the several domains and layers. In principle, they will work together with the business application developers.

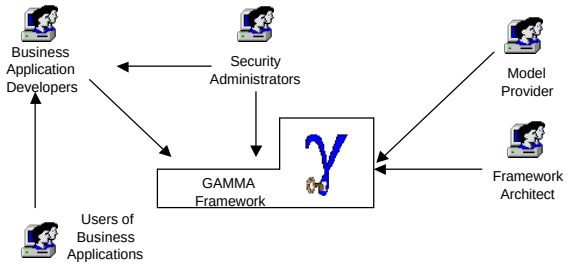

Figure 3 Target groups of the framework.

This categorization is not intended to be exclusive; it is a categorization of interactions rather than of users. Moreover many users will belong to several groups.

2.1. Related Work

Within existing solutions for distributed authorization (compare [4]), we especially concentrated on solutions that support application developers to implement distributed software applications. Furthermore, we looked at existing architectures (e.g. Microsoft .NET, Sun's Java 2 Platform) and compared them with respect to security features according to our requirements.

2.1.1. Distributed Authorization Systems. SESAME [5] provides an infrastructure for authentication, authorization and access control as well as auditing. This distributed security system is based upon Kerberos and provides role-based access control. SESAME does not allow customizing or changing access control models to special application requirements. Furthermore, multiple concurrent access control models cannot be applied at a

time. Summarizing, SESAME is a distributed security system and has not especially been developed for supporting application development by reusable components.

The Adage system [6] provides authentication, authorization and access control as well as auditing in distributed environments. Adage offers a user-centered expression and enforcement of security policies and enables integration in systems due to its modular architecture. However, application developers have limited support by Adage's API. Furthermore, access control models other than RBAC can be defined but require substantial effort to provide them. Nevertheless, the Adage architecture gives important input due to its modular structure of authorization components that enable flexible and adaptable distributed authorization.

2.1.2. Authorization Systems for the Java Platform. There are a number of systems that actively support application developers by providing a framework, class-library or API that can be used instantly when designing and implementing Java applications.

JSEF [7] is a security framework that offers authorization and access control for system-wide security policy maintenance. JSEF especially addresses the security of mobile code. JSEF offers role-based access control including a rich set of possible constraints. Additive and subtractive hierarchies of user groups are provided as well as global and local security policies that are defined using XML. However, JSEF is concentrating on protecting mobile code. Authentication and auditing is based on the underlying Java platform and not part of JSEF. Furthermore, multiple access control models are not supported. Nevertheless, JSEF has interesting ideas concerning a system-wide security policy that is able to merge local and global policy settings.

Kava [8] is a security infrastructure that uses meta-object protocols to provide flexible and fine-grained control over the execution of components. By developing a meta-level security architecture various real-world security models can be realized for Java. Kava inserts security checks directly into compiled code, which prevents modifications to existing code. The meta-object protocol is based on a reflexive extension to the Java runtime that gives the control over the behavior of components. The approach uses byte-code transformation so that the meta-object protocol has control over the components executed in a meta-layer. Kava comes with an expressiveness enforcement of its security mechanisms and supports multiple access control models. However, the transformation of the compiled code is not transparent to the developer and might result in unforeseeable behavior of the application. Furthermore, authentication and auditing is not supported by Kava itself since it concentrates on authorization.

2.1.3 Programming Environments. We investigated today's most important programming environments, namely, Sun's Java 2 platform [9], [10] and Microsoft's .NET environment [11].

Java's best-known security feature is the sandbox model. Code is executed in a protected environment and is not allowed to access resources outside this environment. Furthermore, code is categorized according to its origin into so-called protection domains based on digital signatures. Privileges can be assigned to these protection domains, allowing local code to access more resources and restrict mobile code. However, Java does not support security at a higher logical level by itself. More complex mechanisms are provided as add-on libraries (e.g. the Java Authentication and Authorization Service (JAAS)) that neatly integrate into the Java environment. Nevertheless, adequate security models (a variant of RBAC) are only provided within the Java 2 Enterprise Edition (J2EE), the Standard Edition (J2SE) lacks of these features.

Microsoft's .NET environment offers a wide variety of security features. The evidence-based security evaluates the code's privileges at runtime. The privileges result from digital signed code, the identity of the software developer or the code's origin. .NET offers a role-based access control system and security mechanisms such as Kerberos authentication or various cryptographic algorithms. Within the .NET framework, security is realized in the business objects itself. These objects are extended with code that verifies access. Developers have to perform checks based on the role of their future users. This results in inflexibility since roles have to be defined before implementation is done and cannot be changed later on without modification within the code. Furthermore, reuse of objects is more difficult since security requirements and roles will vary from application to application.

Generally, Java and .NET are programming environments that offer strong security mechanisms (such as cryptographic components) at a lower logical level. Mechanisms at a higher logical level are missing (J2SE), require a special architecture (J2EE) or have to be realized directly in the business objects.

2.1.4. Our Approach. Our approach tries to relieve the application developer of the burden to think about security issues when implementing his/her business objects. The application itself should be security-aware; adoptions to new security requirements should be possible without having to recode or modify the

application. Our approach separates the authorization model from the application. Objects themselves do not know much about security; they are protected by the environment. The protection settings can be modified according to the actual security needs and objects can be reused in any other kind of application. Furthermore, our approach does not require any special infrastructure and can be realized on various platforms and architectures. Finally, none of the approaches mentioned in this section allows to flexibly restrict authorization by constraints as realized within our approach.

3. The Framework Components

In this section we provide an overview of the different framework components, their functionality, and their relationships to other components.

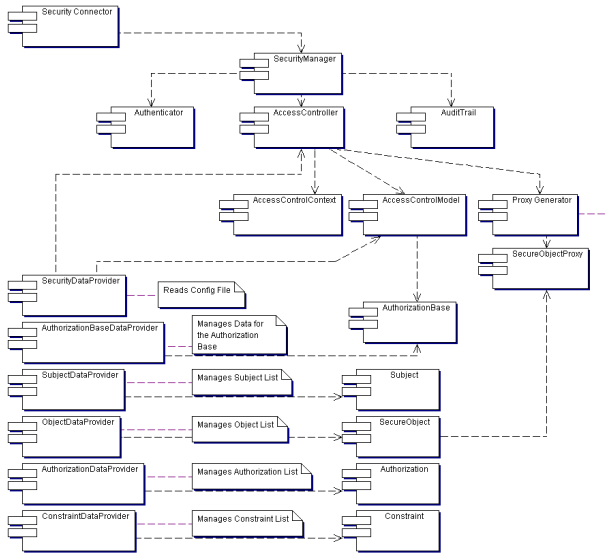

Figure 4 Overview of framework components.

The following sub-sections explain the framework components grouped by their functionality. For the overall picture compare Figure 4.

3.1. Main Components

For providing high-level security mechanisms the framework offers one central coordination component (i.e. security manager) and three specific components corresponding to authentication, access control, and auditing. Within this work, the focus is on access control components.

3.1.1. Security Manager. The *SecurityManager* component controls all other security components of the framework. The application posts requests to the security manager, which dispatches these requests to the corresponding components. Only the security manager knows which components are initialized and running. Thus, it is the central information point for any application. In particular, the security manager handles the interaction between the authentication, access control and auditing components of the framework. However, each machine serving a distributed application may have its own security manager adhering to a centrally defined common security policy. This allows addressing performance issues in distributed environments.

3.1.2. Authentication. The *Authenticator* component is responsible for ensuring correct authentication upon which further access controls will be based. The authenticator checks the subject's identity based upon an identifier. The framework does not require a special kind of identifier, thus the software developer can use various authentication mechanisms. It is the task of the authenticator to validate the identifier according to the method used. The framework currently offers password-based authentication, which will be extended in future releases.

3.1.3. Access Controller. The *AccessController* component is responsible for controlling access to objects according to a particular access control model and based on a valid authentication. This component together with its required infrastructure components are described in detail within sub-section 3.2.

3.1.4. Auditing. A security framework has to be able to track security relevant activities. Thus, the framework provides a flexible audit trail mechanism that gathers messages from all other components and optionally dispatches them to multiple output media. The auditing system allows online filtering of messages for each output media. This can be used for printing critical messages directly on the screen and all other information in a database, for instance. Each output media requires an audit handler that is aware of the media's specialties. It is also feasible to provide a special audit handler that performs online analysis of the audit messages and builds the base for an intrusion detection system.

3.2. Access Control Components

We now want to explain those components in more detail that are related to access control.

3.2.1. Access Control Model. The *AccessControlModel* component is the abstract base class for a concrete access control model. It collects subjects, objects, authorizations,

and constraints from data provider components and transfers them into the authorization base. The model is aware of an underlying closure assumption (open or closed world) when performing access validation, which affects the way that final access decisions are made.

When the access controller contacts the access control model for handling an access request, the model generates a search pattern that contains the requesting subject and the object to which access is requested. It then forwards the pattern to the model's authorization base where rules are searched and analyzed. Depending on the rules one of the following results is returned to the access controller:

- *True*: if a rule is found and access is granted, i.e. the subject has a corresponding permission on the object
- *False*: if a rule is found and access is denied, i.e. a prohibition denies access from the subject to the object
- *Weak True*: if there is no rule found in an open world assumption
- *Weak False*: if there is no rule found in a closed world assumption

Finally, the model is responsible for model specific tasks that are often realized in additional helper classes (such as role activation, ownership, or delegation of access rights).

In our framework we currently support two types of access rules. General rules are assigned to object instances, including all data within the object and its methods. Authorizations on this level grant for example access to data stored within the object. The second type of rules are specific rules. These rules are assigned to a certain object's method. The rule applies only to this method, all other methods of an object are unaffected.

3.2.2. Access Control Context. The *AccessControl-Context* is an abstract component that captures meta data needed by various other components when making access control decisions. The particular structure has to be defined as required by the access control model.

3.2.3. Autorization Base. The *AuthorizationBase* component realizes a storage holding tuples with references to a subject, an object, optionally a class or method, a certain authorization and an optional list of constraints. Each tuple is a rule that describes how a subject may access an object, a class or a method based on an authorization and constraints. When the defined subject requests access to an object, all matching rules are considered and the authorizations' as well as constraints' *checkAccess* methods are called.

Each access control model has its own authorization base. Thus, different types of access control models might have to realize an adapted authorization base, in order to capture its particular functionality.

3.3. Infrastructure Components

The components described within this sub-section are required by any of the main components especially by the access control components in order to enforce their tasks.

3.3.1. Subject. The *Subject* component realizes actors and entities of the system, such as persons, processes or access control model specific entities (e.g. roles), each represented by an adequate sub-class that possible has to be derived and implemented.

When a subject wants to execute a method on an object, which requires access to another object, basically two options can be followed:

- the object becomes a subject and access control is done by verifying the corresponding object's access rights, or
- the first object is calling the method of the second object on the behalf of the subject. Therefore, the subject has to have sufficient access rights itself to fulfill the task (transitive access).

When accessing an object transitively, the subject of the first object is used for evaluating access. This subject, as a part of the request parameters, is stored in the access control context. When access to the second object is requested, the access controller can retrieve the subject from the access control context and check whether or not the subject is allowed to call the method on the second object. In that case the transitively accessed objects trust the security framework to correctly authenticate the originating subject.

3.3.2. Secure Object. The *SecureObject* component is the base class for all objects, which need to be protected within the framework. In order to ensure enforcement, it is necessary that a client does not obtain a direct reference to the object. By hiding the constructor, only a privileged component can create an instance of a secure object. This component returns a *SecureObjectWrapper* as a proxy instead of the object itself. The object can only be accessed by the proxy object, which guarantees that security checks are called before the object is accessed.

In order to use secure object wrappers, there has to be a central component that is aware of generating these components. The *ProxyGenerator* component is aware of the existing objects and can create secure object wrappers using introspection. In fact, the Java language offers a ready-to-use component that realizes this proxy generator. In other environments, this component has to be created using the language's runtime information.

Objects are identified via a unique id within the framework. This id is computed from the object's full-qualified class name and an instance identifier. The prefixed class name allows an easier lookup of the object and administration within the data storages. However, the algorithm for computing the object's id is adaptable according to personal needs.

3.3.3. Authorizations. The *Authorization* component realizes the type of access rights onto a resource. Authorizations are logically separated from the access control model to increase flexibility. Thus, authorizations are implemented by the business application developers and have a defined meaning for a given secure object. As a consequence, each authorization component has a *checkAccess* method. This method is called when access validation is done. The framework also offers specialized components for positive (permissions) and negative (prohibitions) authorizations. The usage of the specific type of authorization depends on the *closure assumption* maintained by the access control model.

Authorizations can be defined either on object level or for specific methods of an object. Thus, the developer of an authorization must additionally take care of different meanings of a certain authorization (e.g. an execute permission at object level allows the invocation of all methods whereas at method level it allows only the execution of a certain method). This implies also that some authorizations are applicable only to object level or to method level.

3.3.4. Constraints. The *Constraint* component allows to further restrict authorization within the system in a flexible way. Some constraints are specific to particular access control models (e.g. separation-of-duty). These specific constraints influence only actions and tasks of the access control model and are defined and implemented by the model provider. Other constraints are independent of the current active access control model and influence the application as a whole. They have to be defined and implemented by the framework architect. Examples include *location* constraints (e.g. logins are only allowed from specific IP addresses), *time* constraints (e.g. logins are allowed from 8am to 5pm).

Similar to the authorization component, the constraint component offers a *checkAccess* method that is able to evaluate access.

3.4. Data Provider Components

The data provider components realize the interoperation with underlying backend systems. Each infrastructure component mentioned in section 3.3 (i.e. subject, secure object, authorization and constraint) needs to have a specific data provider, which is able to read and write from and to persistent storage (compare

Figure 5). A so-called *security data provider* manages the configuration of the framework and the relation between an application and it's specific data providers.

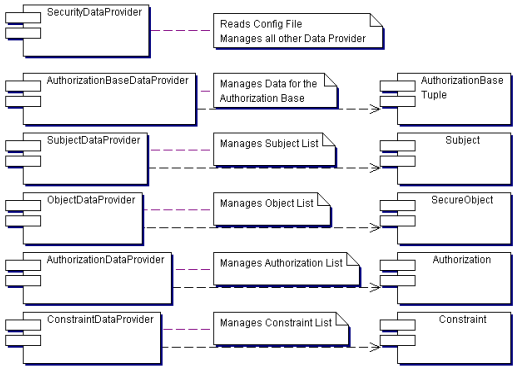

Figure 5 Data provider components.

This security data provider handles the framework's configuration by interacting with an XML-based storage. The configuration file points to various other data providers that are able to communicate with various objects storages (e.g. LDAP, database). The security data provider coordinates these data providers that are necessary for the establishment of the authorization base.

Each security component has its own data provider which allows the connection to different systems (e.g. users can be taken from the operating systems whereas objects are stored within a database). Furthermore, data providers take care of the different needs of the security model (e.g. a DAC model merely needs users, an RBAC model needs users and roles), thus different data providers can be specified for different models. In general, each model must reference to a subject data provider that manages the model's subject, an object data provider for the objects to be protected, an authorization data provider pointing to the various authorizations available, a constraint data provider supplying constraints and the authorization base data provider that manages the authorization statements in the form of subject, object, authorization and constraint tuples.

If a tuple in the authorization base cannot be resolved due to missing entities, the tuple is removed and an auditing message is generated.

4. Realization Issues

We now want to discuss particular realization issues encountered when implementing the framework. Currently, GAMMA is available as beta-release implemented in Java. As mentioned before, we want to

realize the platform and architecture independent design also within the .NET framework in future.

4.1. Canonical Access Control Mechanism

For providing generic access controls an extremely flexible way of enforcing access controls has to be found. The basic assumption is, that access controls can be stated in terms of subjects accessing protected objects. Particular access control models have to implement particular authorization components, stating the semantics of authorization within that model. Furthermore, access control models can make use of constraints that capsulate either model specific authorization conditions (e.g. separation of duty) or general authorization conditions like time or location constraints, for instance. Figure 6 illustrates the particular steps of the canonical access control mechanism implemented in GAMMA.

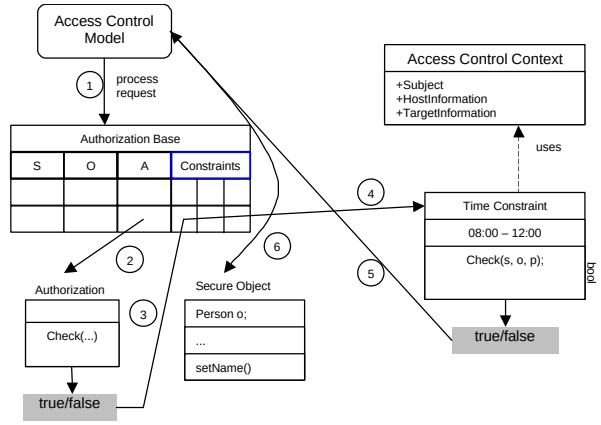

Figure 6 Canonical access control mechanism.

In general, a subject wants to access a protected object in a certain way. The requested operation on the object defines authorizations that are necessary in order to fulfill the task. The way how to decide whether an access is granted or not is determined by the access control model, the authorization objects defined for that model, and the constraint objects assigned to the authorizations.

First the access controller receives a request from an authenticated subject for a certain operation on a protected object, which has been mediated by the security manager. The request is passed to all active access control models according to a particular order specified within a configuration file. Further details on using multiple concurrent access control models are discussed in section 4.2.

Each access control model searches for a subject/object combination in the authorization base (1), which matches the request. The search process returns a list of matching authorization rules that are defined for the subject/object combination. Each authorization is explicitly checked by invoking the authorization's *checkAccess*-method (2). However, there is the possibility of defining additional constraints that further restrict a specified access operation to a protected object. Thus, each constraint that is defined for the particular authorization is evaluated. Again, the decision is delegated by invoking the *checkAccess*-method of the particular constraint (3/4). When both, the authorization and any constraint suggest granting access, the access control model either reports a positive (access is granted) or negative (access is denied) result (5) depending on the closure assumption of the model (open/close world). This result is then returned to the security manager via the access controller, which finally gives or prevents access to the requested object (6).

This possibility may lead to conflicts requiring each model to have a conflict resolution strategy. We propose and implemented a strategy, where – in that order – specific rules (assigned to an object's method) win over general ones (defined on object- or class level), prohibitions over permissions, and any rule wins over the closure assumption, which in anyway regulates the case of having no applicable authorization rule. Consequently, it may happen that a specific permission wins over a general prohibition. For instance, a bank secretary may be generally prohibited to access individual bank accounts retrieving the name or other data. However, the secretary may be specifically allowed to retrieve the total amount of money stored in a certain account by means of invoking a certain method (e.g. getAmount) in order to perform statistical measurements. However, it remains in the hands of the model provider to define the conflict resolution strategy that is most appropriate to the particular application's security requirements.

Currently, a traditional DAC approach as well as an RBAC approach according to NIST standardization efforts (compare [12]) has been realized based on the aforementioned canonical access control mechanism. However, we plan to realize multi-level approaches as one of the next steps in the project in order to evaluate the spectrum of applicability of our canonical access control mechanism.

4.2. Multiple Concurrent Access Control Models

When using multiple access control models, it is probable that different models have different meanings concerning access privileges. Thus a strategy to resolve such conflicts is required. An example of concurrent access control models is given in section 5.

Any access control model that should be used is specified within a framework configuration file. The

sequence in which they are listed defines the domination of the models. When access is checked, the access controller contacts the active models, starting with the first one. Each model can return four values, which indicate either strong or weak results. If the model returns a strong result, the access controller accepts this result and returns it to the security manager, thus a strong result is mandatory. A weak result indicates that the current model cannot make a proper choice but returns a proposal based on its closure assumption. In this case the access controller contacts the next active access control model as long as a strong result is returned. If each model returns weak results indicating that no model is able to make a proper choice, the first weak result is returned.

This mechanism ensures that a result is found and that the access control mechanism pays attention to the domination order of the active access control models. If a more dominant model is able to make an obligatory choice, this choice is taken irrespectively of all other less dominant models. However, if a model cannot make a proper choice but a less dominant can, the most proper choice is taken and returned to the security manager.

4.3. Security Enforcement

A proxy controls access to an object with the help of a prefixed representative object [13]. Access to the real object is only possibly through the proxy. GAMMA protects data objects that contain sensitive information by automatically generating proxy objects and returns them instead of the real object. Proxies offer the very same routines as real objects, thus the client does not recognize any differences. Accessing these objects via the proxy directly invokes the access control mechanism. Since the real objects are kept in a separate space and can only be accessed through proxies, an application cannot circumvent or bypass the access control mechanism. In fact, a client will never obtain a direct reference but a proxy that seems to be the real object.

4.4. Reflection

To provide GAMMA's flexibility, a mechanism is necessary to introspect software objects at runtime. This is necessary to generate proxies but also to load user-defined framework extensions (e.g. new access control models, data providers). The current implementation of GAMMA is realized using the Java language. Java offers introspection – called reflection. However, GAMMA's concept can be implemented in any other language for any other architecture (e.g. Microsoft .NET) where introspection facilities can be realized.

5. Example: Time Management

In this section we present an example application that is simple yet incorporates comprehensive authorization requirements to demonstrate the benefits of the presented component framework. The sample application has been implemented as vision demonstrator within the GAMMA project.

5.1. Authorization Requirements

Figure 7 illustrates the authorization requirements of the time management application.

During a month employees record their project hours into timetable objects. Each employee is owner of his/her monthly timetable and shall thus have full control over the object. The employee shall furthermore be allowed to pass access to his/her timetable object to third persons, if (s)he wants to (REQ1). Additionally, project managers shall have read access to their employee's timetables (REQ2). However, at the end of each month employees shall finish their timetables and loose write access over them. Rights granted to individual third persons shall be revoked. Project managers shall have the ability to finalize their project reports using the timetables of their project members. Furthermore, full control of any timetable shall be passed to a secretary, who balances accounts with customers (REQ3).

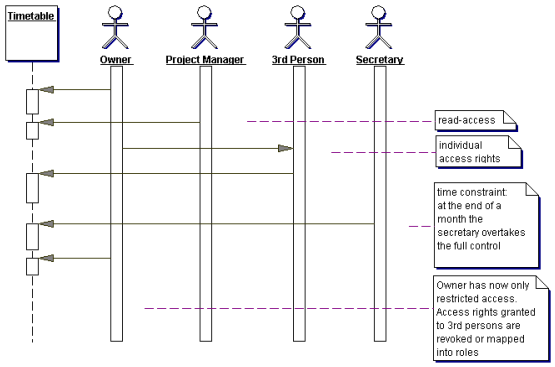

Figure 7 Authorization requirements.

In this example, REQ1 can be realized using a discretionary access control (DAC) model. REQ2 can be best realized by using a role-based access control (RBAC) model, where a role "Project Manager" has always the right to read the timetables of his/her employees. In the case of combining DAC and RBAC mechanisms it is necessary to define that RBAC mechanisms shall be stronger than DAC mechanisms, since an employee shall not be able to revoke read-access to his/her timetables from his/her project-manager. According to REQ3 a change to the authorization scheme is caused by a

temporal event. At the end of each month, the owners of timetable objects lose their full control automatically and thus delegate access rights to dedicated roles (i.e. project manager and secretary).

The combination of DAC and RBAC is changed into a single RBAC model. The role "Secretary" now has the full control over the timetable objects and is the only subject who is able to do modifications. The owner and the project manager still have read access or - when granted by the secretary - also restricted write access. Subjects that had access to the object before - granted by the owner - now lose their access rights to the object.

5.2. Realization using Multiple Access Control Models

This example also highlights the ability of GAMMA to work with two access control models at the same time, i.e. a DAC and an RBAC model, both using a closed world assumption.

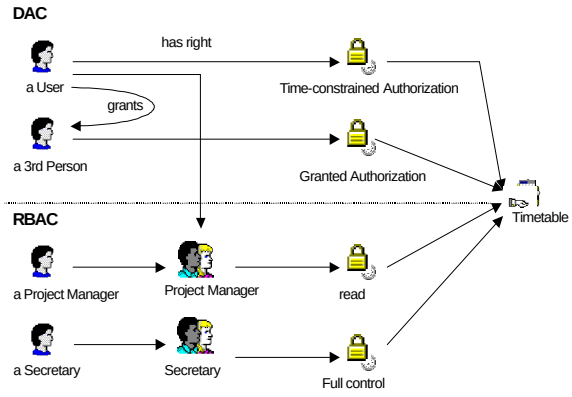

Figure 8 Concurrent DAC and RBAC models.

Figure 8 illustrates the schematic combination of the two access control models. At the beginning of each month, an employee creates his/her timetable, which is an instance of a class derived from the secure object component. The employee automatically receives a special authorization "ownership", which grants full access to the timetable. Additionally, a time-constraint is associated with the ownership authorization (indicated by a watch symbol) that restricts the employee's ownership-privilege to the current month. When granting access to third persons additional time-constraints are automatically created also restricting the delegated access rights to the current month.

At the end of the month, when the timetable is given to the secretary, the ownership privilege as well as any right granted to a third person is automatically disabled due to the time constraints.

At the same time some other persons have access to the timetable based on appropriate roles. These are, for example, the project manager or the secretary. Roles are assigned to users. Additionally, authorizations are assigned to roles. However, these authorizations also have time constraints – in fact exactly those constraints that are complement to the DAC authorizations' time constraints.

Using this approach the validity period of access control models can be regulated. During the month only rules specified in the DAC's authorization base are valid. At the end of the month, all these rules become invalid since their time-constraints do not allow access anymore. However, the rules of the RBAC model become valid.

5.3. Used/Extended Framework Components

The authorization requirements and the realization approach mentioned above can be realized with GAMMA using and/or extending particular components of the framework.

First of all, a timetable business object is created by business application developers, which needs to be protected and has thus to be derived from a secure object. Then, model providers have to derive two concrete access control models, namely, DAC and RBAC that will be specialized for that particular application. Framework architects may later on decide to include the resulting access control model with the framework for future usage. Application developers and model providers negotiate and derive the particular subjects to be used in this application, i.e. employees (as users) and roles. Furthermore, they define concrete authorizations applicable for that scenario such as *ownership* (full access and the right to grant access to third persons), *read permission* (allowing a subject to view the contents of a timetable), *write permission* (allowing a subject to modify the contents of a timetable), and *delete permission* (allowing a subject to delete a timetable). Additionally, time constraints have to be derived that are automatically associated to authorizations in order to realize the temporal switch of access control models. Finally, model provides have to realize concrete data provider objects if not yet available for the particular backend systems.

6. Conclusions and Future Work

In this paper we have presented GAMMA, a security framework supporting the development of multi-tier applications. It contains a variety of components offering a range of high-level security mechanisms including different access control models that can be used concurrently and the support of arbitrary constraints.

Based on the experience of having used GAMMA for several applications we have identified the following major issues that we plan to address in future:

First, we strive to realize the framework for the .NET programming environment, which does not yet offer the convenient enforcement. This shall be a straight forward task due to the language and architecture independent design of the GAMMA framework.

Second, we will implement additional forms of authentication, since the GAMMA prototype currently only provides an implementation for password-based authentication. In particular, we will concentrate on challenge-response methods.

Third, since multi-level approaches can be useful in some situations we will also implement models such as the Bell-LaPadula model. We think that one reason why MLS is not widely used is the lack of support in today's developing environments. That said, we will also be able to evaluate the spectrum of applicability of our canonical access control mechanism.

And finally, several standards bodies, including OASIS (Organization for the Advancement of Structured Information Standards), IETF (Internet Engineering Task Force) and W3C (World Wide Web Consortium), have proposed XML-based security standards. The most relevant with respect to authorization and access controls are SAML (Security Assertion Markup Language) and XACML (eXtensible Access Control Markup Language), both driven by OASIS technical committees. SAML is an XML-based framework for exchanging information about authentication acts performed by subjects, attributes of subjects, and authorization decisions about whether subjects are allowed to access certain resources or not. The specification is currently under consideration for becoming an official OASIS standard. XACML, on the other hand, defines a core schema and a corresponding namespace for the expression of authorization policies in XML against objects that are themselves identified in XML. XACML is currently an OASIS working draft. At the time starting to develop GAMMA, these standards were not available or at least not stable enough to consider integration into GAMMA. However, our architecture allows realizing dedicated security data provider for SAML or XACML, for instance, which we plan to do, as soon as stable specifications are available.

References

[1] J.E. Dobson and B. Randell B, "Building Reliable Secure Computing Systems out of Unreliable Insecure Components", *Proc. 17th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, Louisiana, USA., Dec. 10th –14th, 2001.

[2] M. Carey, H. Garcia-Molina, J. Hamilton, H. Pirahesh, and B. Thuraisingham, "Where are Our Promising Research Directions: Database Server, Middleware, or Applications?", Panel Discussion: *Proc. 18th Int. IEEE Conf. on Data Engineering (ICDE)*, San Jose, CA, USA, Feb. 26th – March 1st, 2002.

[3] W. Essmayr, S. Probst, and E. Weippl, "Role-based Access Controls: Status, Dissemination, and Prospects for Generic Security Mechanisms", *to appear: Int. Journal of Electronic Commerce Research*, Kluwer Academic Publishers.

[4] W. Essmayr, S. Probst, and E. Weippl, "A Comparison of Distributed Authorization Solutions", *Proc. 3rd Int. Conference on Information Integration and Web-based Applications & Services (IIWAS)*, Linz, Austria, Sep. 10th-12th, 2001.

[5] P. Ashley, and M. Vandequauver, "Intranet Security - The SESAME Approach", Kluwer Academic Publishing, 1998.

[6] M.E. Zurko, R. Simon, and T. Sanfilippo, "A User-Centered, Modular Authorization Service Built on an RBAC Foundation", *Proc. IEEE Symposium on Security and Privacy*, Berkley, CA, USA, May 1999.

[7] M. Hauswirth, C. Kerer, and R. Kurmanowytsch, "A flexible and extensible security framework for Java code", *Proc. 9th International World Wide Web Conference*, Amsterdam, May 2000.

[8] I. Welch, and R. Stroud, "Supporting Real World Security Models in Java", *Proc. 7th IEEE Workshop on Future Trends in Distributed Computing Systems*, Tunisia, South Africa, December 1999.

[9] Sun Microsystems, "Java Security Architecture", http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-specTOC.fm.html (last accessed on May 31, 2002).

[10] E. Jendrock, S. Bodoff, D. Green, K. Haase, M. Pawlan, and B. Stearns, *The J2EE Tutorial*, ISBN 0-201-79168-4, Addison Wesley, 2002.

[11] Foundstone Inc., CORE Security Technologies, "Security in the Microsoft® .NET Framework", http://www.foundstone.com/pdf/dotnet-security-framework.pdf (last accessed on May 31, 2002).

[12] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST Model for Role-Based Access Control: Towards A Unified Standard", *Proc. 5th ACM Workshop on Role-Based Access Control*, July 2000.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, ISBN 0-201-63361-2, Addison-Wesley, 1995.