A Flexible Security Architecture for the EJB Framework

Frank Kohmann[1], Michael Weber[2], Achim Botz[1]

[1] TPS Labs AG, Balanstr 49, D-81541 München
{frank.kohmann achim.botz}@tps-labs.com
[2] Abteilung Verteilte Systeme, Universiät Ulm, D-89069 Ulm
weber@informatik.uni-ulm.de

Since the Enterprise JavaBeans (EJB) specification 1.0 has been released in spring 1998, a great interest has been shown in this new technology. It is the first approach that defines a server-side component model for the Java platform. The EJB specification already has achieved a high grade of acceptance and will obtain an important role in the development of distributed applications in the future. Nevertheless, since EJB is a young technology it still has to struggle with several problems. One of these is security management. The issues concerning security that are covered by the specification are very basic. However, the distributed nature of EJB makes a sophisticated approach in regard to security indispensable. Due to this necessity, a flexible security architecture for the EJB framework is proposed within this work.

1 Introduction

The Enterprise JavaBeans (EJB) framework [1] defines a component model for the development and deployment of Java applications, focusing on the server-side. EJB makes it possible to develop server components and plug them together to entire server applications. The EJB framework provides low-level services, such as transaction management or object persistence. By providing such services at the framework level, it becomes possible to implement components, and in the end server applications, which are free of code concerning these services. Therefore, distributed applications become easier to develop, more flexible and more scalable. A separation between the business logic and the distribution services is gained. This new philosophy introduces an important alternative to the traditional client-server architecture. Since the specification has been released, a great interest in this new concept of distributed programming has been shown, not only from application server vendors, but also from server application and client application providers.

However, EJB is a young technology and still has to struggle with several problems. One of these is security management. The issues concerning security that are covered by the specification are very basic. However, the distributed nature of EJB makes a sophisticated approach in regard to security indispensable. Due to this necessity, a flexible security architecture, based on Java 2, for the EJB framework is

proposed within this paper. A good understanding of EJB is presumed. For an introduction see [2] and [3].

The next chapter outlines the most serious drawbacks of the EJB specification 1.0. The first part of chapter 3 gives a general overview of the introduced security architecture, which then is described in more detail in the rest of that chapter. The final chapters are related work and a conclusion.

2 Security Concepts of EJB Specification 1.0 and its Drawbacks

The specification is not very detailed about security management. By leaving such details unspecified the development of code that is portable between different server or container implementations is hindered. This is obvious when looking at authentication, which is not specified at all.

The only feature concerning security that is part of the specification is authorization. The deployment descriptor allows associating an entire bean or a bean's method with one or more identities. At runtime, these specified identities are permitted to call the bean or the bean's methods while the access is denied to others.

When authorization is solved the associated problem of delegation must be taken into account. This is a broad field with a variety of different possible strategies. When looking at a chain of subsequent bean invocations, the main question is according to which privileges a bean should execute. Should the bean's privileges be used or the caller's rights passed to the bean in some manner? The only approach the specification provides, is the possibility to define an identity (the so-called RunAsIdentity) at deployment time under which the bean's methods always have to be executed.

From these basic concepts of security management that are defined by the current specification result the following serious drawbacks:

- No interfaces for server independent authentication are provided.
- Authorization can only be achieved through an ACL like approach (list of identities for a bean or bean's method). There is no possibility to implement other authorization strategies like, for example, label-based authorization.
- Access control decisions cannot be made on instances. This is essential for entity beans.
- There exists no concept of delegation.

3 A Possible EJB Security Architecture

The scope of this work is the introduction of an abstract security architecture with the according interfaces and a description of its behavior. No concrete implementation of a security solution is proposed to maintain the flexibility of implementing the architecture with different technologies. The actual specification does not provide the flexibility to plug in such a security solution. The major reason is the lack of specified interfaces and rules of interactions between container and server. Therefore, changes

to the specification have been made in regard to the deployment descriptor and the objects EJBMetaData and EJBContext. Furthermore, additional responsibilities have been added for the container.

The security architecture provides a secure environment for the beans. The granularity the architecture has to deal with is on one hand the differentiation into beans and on the other hand the distinction of bean's methods. When dealing with security at method level, all methods of a bean instance are relevant. This not only includes the methods the bean programmer provides but also the methods inherited from EJBHome and EJBObject, like create() for example.

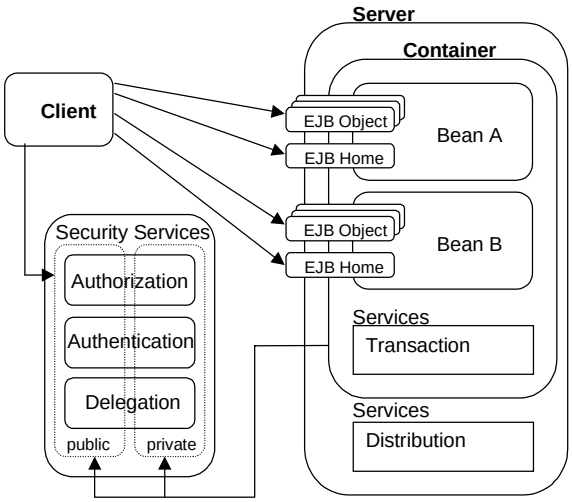

Fig. 1. The changed EJB Architecture.

The main idea is to provide a domain concept for the EJB framework that is similar to the one of the CORBA Security Service [4]. The notion of the domain is introduced to enable a better structuring of the different functionality the architecture has to meet. A domain implements a specific security policy according to a certain technology and provides an interface to access its services. A domain could, for example, implement password authentication or ACL-based authorization. Three different types of domains are introduced: authentication, authorization and delegation domains. Each bean has to be member of at least one domain of each type, but may be in multiple domains of the same type. If the latter is the case, one domain has to be marked as default domain. A domain provides two types of methods: Methods that are accessible to everybody, in particular to the client, and methods that are called by the container. The first type of methods provides functionality needed by the client like getting information about the domain or authentication methods. The methods called by the container are in charge to fulfill the policy of the domain. This splitting of the domain is made for security reasons. Both the client and the container need to access the domain but each with different intention. The client, for example, should not be able to gain information on details of the authorization algorithm.

All new functionality that is provided by this architecture (Figure 1) is plugged into the container. However, this functionality is not implemented within the container classes, but within the domain implementation. The container is only responsible to delegate security related tasks to the appropriate domain implementation at certain times. These domain implementations form the pluggable security service. Like the naming service it is independent of the EJB architecture. An EJB server is free to use any implementation of a security service. This could be a proprietary service as well as a service of another vendor.

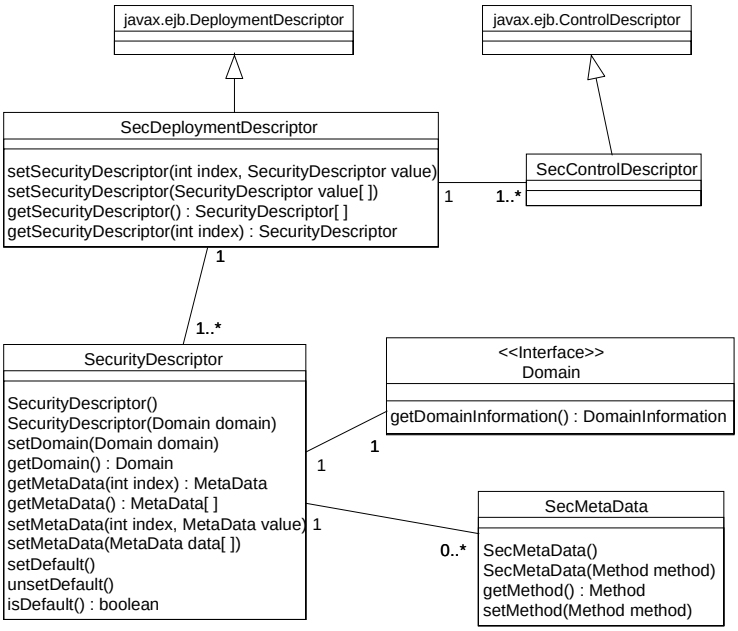

Fig. 2. Relation between domain and deployment descriptor. SecDeploymentDescriptor additionally has all methods of DeploymentDescriptor. SecControlDescriptor is the same as ControlDescriptor but without maintaining RunAsIdentity and RunAsMode.

The great advantage of separating the implementation of the security functionality and the container implementation is the flexibility gained. If there exists a variety of different container implementations, it would be very cumbersome, needing to make changes in the code of every container one wanted to enhance with the ability to deal with security. Moreover, for every change in the policy the container would have to be modified. The third reason for this architecture is the complexity of the security service's implementation. Authentication, authorization or delegation and their possible mapping to existing legacy or operating systems can be very complex. Therefore, in addition to the roles defined in the specification, the role of the security provider is defined. The security provider is responsible for the implementation of the security domains. To achieve this goal the specification needs to be changed in the

way that the container supports the domain interfaces introduced in this work. These interfaces then provide the possibility to plug in the different implementations of the security domains.

The information on domain memberships of the beans as well as additional data required by the domain to implement the policy is maintained in a modified deployment descriptor (Figure 2). A security descriptor is introduced to be able to bring a domain in relation with a bean. Each security descriptor represents the membership of the related bean in the appropriate domain. Each domain might need different information connected to the bean or the bean's methods in order to fulfill the policy. In the case of an ACL-based authorization approach this might be the identities that are allowed to invoke the different methods while in a label-based solution only the value of the label is needed. To solve this issue the SecMetaData object has been additionally introduced. This object is like the domain itself provided by the security provider and is used to bring a bean or a bean's method in relation with domain specific data.

3.1 Authentication

Authentication is introduced for beans and clients. Since neither the container nor the server is concerned with security issues, authentication is not provided for these two objects. It is presumed that they are trusted and run with system level privileges.

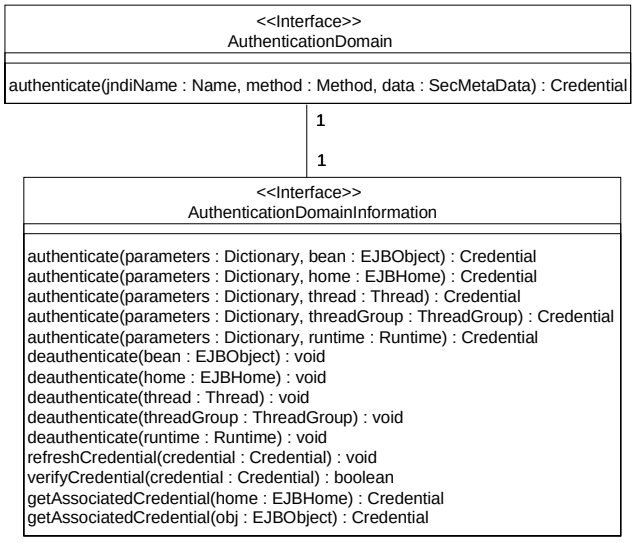

Fig. 3. Interfaces representing the authentication domain.

The main task of an authentication domain (Figure 3) is to establish a security context to be used when the authenticated object performs an action. This security context is the credential. It is a domain specific object that ensures the authenticity of the caller. In case of client authentication, the credential also is related to the runtime scope for which it was created. The client has the possibility to set certain information in the credential. If a bean is member of multiple delegation domains, for example, the client can specify here the domain that is to be useed. In addition to the authenticate() methods an authentication domain needs the method verifyCredential() in order to prove the validity of a credential. A refreshCredential() method is important for refreshing a credential without needing to perform the possibly time-expensive authentication procedure. Furthermore, the getAssociatedCredential() methods have to be implemented by every authentication domain. They are needed to ensure the interoperability of the classes provided by the container provider and those of the security provider. Although the stub does not know anything of the credential creation, it needs to propagate it to the container. By calling getAssociatedCredential() it obtains the credential that applies for the current execution context.

Bean Authentication. It is the responsibility of the container to authenticate a bean. This process is completely transparent to the client. It is up to the container whether the authentication takes place before the first invocation of an instance's method or at the time the bean is created.

Whether a method finally executes with the privileges of the credential it has been authenticated with or not, also depends on the delegation strategy used.

Bean authentication is an enhancement of the former possibility to set the RunAsMode or the RunAsIdentity and is necessary for delegation.

Client Authentication. A client can be any kind of Java application including applets or servlets. A bean as well can be a client. In this case, the security context established automatically by the container is overridden by the context created during the explicit client authentication. Client authentication is possible at five different levels:

- Runtime
- ThreadGroup
- Thread
- EJBHome
- EJBObject

The authenticate() methods take as parameter one of these scopes and some domain specific parameters. Latter are passed via a Dictionary object. A password authentication domain, for example, typically needs a password and the user id.

During authentication a credential is established. The credential is used for all invocations made within the scope in which it was created. This idea is straightforward for the authentication at the level of Runtime, ThreadGroup and Thread. If a client authenticates at thread level, for example, all further calls made to the EJB server within this thread are transparently extended with this security context. Calls from other threads are treated as unauthenticated calls. For authentication at

level of EJBHome and EJBObject this means the following: If a credential is created at the level of an EJBHome object it is passed to the container for every method the client invokes on this home object. Additionally the same credential is used for method invocations on all beans created via the authenticated home. In case the client is authenticated only for a special bean, that is at EJBObject level, the resulting credential is used only when invoking methods on that bean.

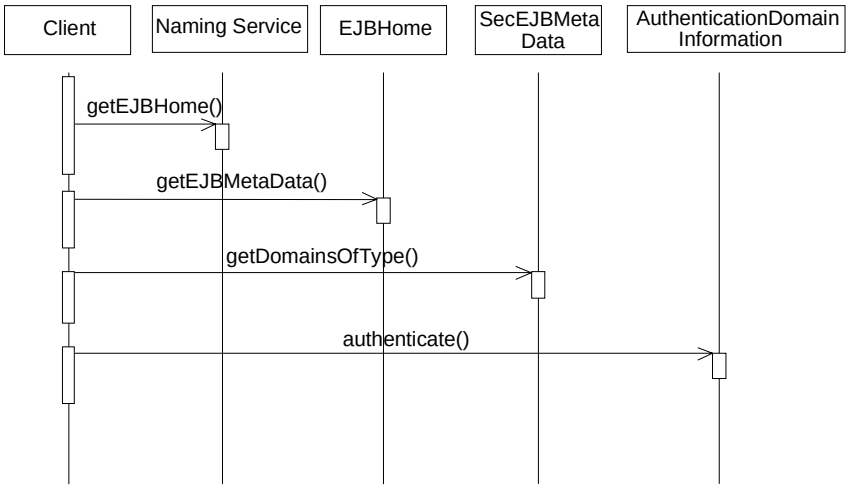

Fig. 4. Process of client authentication. SecEJBMetaData is EJBMetaData enhanced with the possibility to get information about domain memberships. AuthenticationDomainInformation represents the publicly accessible part of the domain.

Now it can happen that the client is authenticated at runtime level as well as at EJBHome level. In this case, the credential that was created in the most specific context is used to propagate the security context of the client to the container.

Authentication at the execution level of Runtime, Thread and ThreadGroup are indispensable for easy application authentication and in particular for multithreaded applications. The levels of EJBHome and EJBObject can be relevant for servlets. Since these objects can be shared among different clients each client request could be handled in a distinct thread context. Thus, a possibility to perform an authentication at a lower level than Thread level is needed. It should be kept in mind though that the performance of an application goes down if finer grained levels of authentication are used, since more objects have to be authenticated.

For the process of the client authentication see figure 4.

3.2 Authorization

The responsibility of the authorization domain (Figure 5) is to check whether the access to a bean's method is allowed or not. Before the container invokes a method on

a bean's instance the authorization domain is consulted by invoking the checkAccess() method. Depending on the result the container can take the appropriate action, either continue or throw an exception.

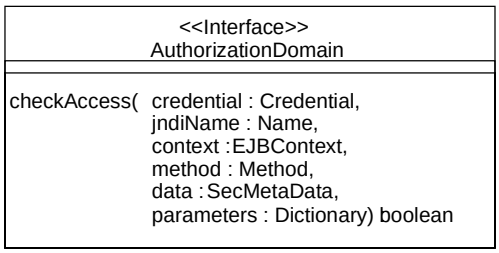

Fig. 5. Authorization Domain interface. As parameters are provided the credential of the caller, the name and the method of the bean that has been called, information on the bean instance, data required by the domain and the parameters the method was in called with.

In order to work correctly, checkAccess() has to be provided with a couple of parameters. Some domain implementations might not need all of them. However, since this is a general interface all possibly needed information is provided.

A parameter that needs special attention is EJBContext. It provides the possibility to access a bean's instance and therefore provides the possibility to make access decisions on instances. This is very useful for entity beans. Checking access at class level is not convenient for sophisticated authorization policies. Some policies might want to deny access to beans due to the value of instance variables. Via the getPrimaryKey() or the getEJBObject() methods of EntityContext, which is derived from EJBContext, information about the instance can be accessed.

The whole authorization process is transparent to the client as well as to the bean. While this is wanted in the most cases, sometimes it might be useful if the bean implementation as well could perform some access control checks. The solution is to provide additional code within the bean itself. At the beginning of a method the bean can obtain the credential of the caller via the EJBContext. This is possible through the getCallerCredential() method that has been appended additionally to getCallerIdentity(). With the help of the credential the additional check can be performed. It is important to state that these checks are always additional to the checks performed by the domain.

3.3 Delegation

The delegation domain (Figure 6) decides in which context an operation of a bean is performed. Like stated before every object of the EJB framework is associated with a credential. When a client invokes a bean or a bean invokes another bean, the delegation domain decides which privileges have to be used by what bean to perform an operation. An easy policy could, for example, simply always use the privileges of the caller not caring about the privileges of the bean that has been called.

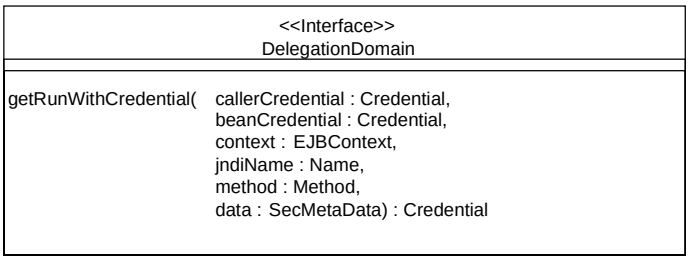

Fig. 6. Delegation Domain interface. As parameters are provided the credential of the caller, the credential of the bean that has been called, the name and the method of the bean that has been called and data required by the domain.

The delegation domain is called via the method getRunWithCredential(). It is called by the container after calling checkAccess() but before invoking the method on the bean's instance. The result of getRunWithCredential() is another credential. This credential represents the security context within which the bean's method is executed.

For an overview of the involved objects and methods upon a client invocation of a bean's method see figure 7.

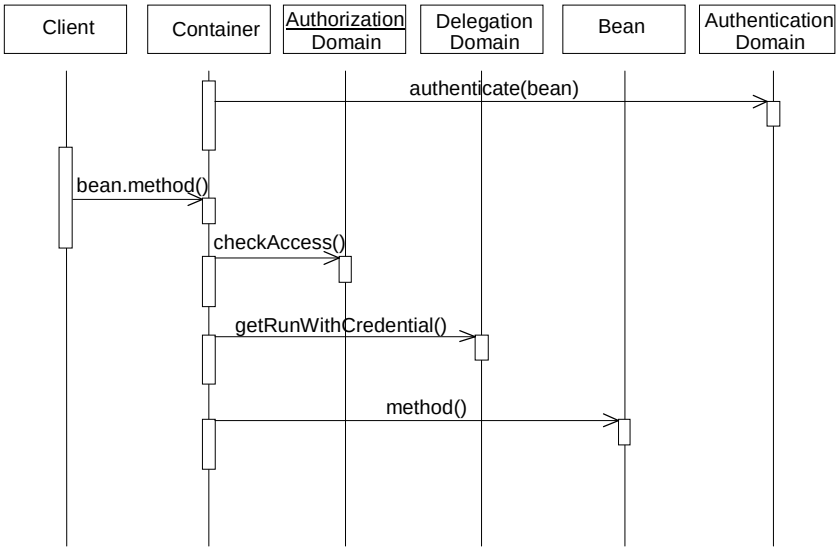

Fig. 7. Process of invoking a method on a bean's instance.

4 Related Work

A very important work to be mentioned is the Java Authentication and Authorization Service (JAAS) that recently has been proposed [5]. It deals with the issue of introducing client authentication and authorization into the Java environment. It is not an API yet, but rather a proposal that is under construction. While some basic approaches of both proposals are similar, like the independence of the implementation details for instance, they differ in several ways. The main difference is that the JAAS approach focuses more on the needs of non-distributed applications while this proposal focuses on the distributed nature of EJB. Nevertheless, it should be thought of combining the different concepts in order to provide a single security architecture for the Java environment in the future.

Our domain concept is similar to the one of the CORBA Security Service. However, the use of the domain concept in conjunction with the EJB framework is different to the envisaged deployment of the CORBA Security Service with the CORBA Component Model.

Further interesting research in regard to Java and security is being performed at the University of Erlangen-Nürnberg [6]. Here the idea of attaching meta-objects to Java references is investigated. This capability approach clearly separates the application code from security policies and is also very flexible. For the implementation of this strategy, the MetaJava system [7] is used, which allows the binding of meta-objects to conventional Java references. The use of this system however, implies the use of a package of native code.

5 Conclusions

A domain concept that is similar to the one of the CORBA Security Service has been developed for the EJB framework. Interfaces for domains have been introduced that can be of the types authentication, authorization and delegation. Each bean has to be member of at least one domain of each type. The implementation of the interfaces provides the realization of a certain policy.

Not only the client and the beans, but also the container are independent of the underlying domain implementation. The container's responsibilities change in the way that it has to make certain calls to the domain at certain times. General interfaces have been introduced to maintain such independence between the domain and container implementations.

Due to the complexity of the security issue, the new role of the security provider has been introduced additionally to the existing roles. The main responsibility of the security provider is the implementation of the domain interfaces that realize a certain policy. This also can include the mapping to existing security services.

The proposed architecture enriches the EJB framework with a flexible possibility to provide server applications with the security they need. At the same time, the beans as well as the client code can be kept independent of the implementation details of the domains. Furthermore, the most important aspect of EJB is maintained: Enabling the

bean provider to focus on the business logic and not needing to care about security issues.

Acknowledgements

This work has been supported by TPS Labs AG (www.tps-labs.com). The authors would especially like to thank Günther Heiss and Ingo Körber for providing helpful feedback and useful discussions.

References

1. Sun Microsystems. Enterprise JavaBeans Specification 1.0., 1998. http://java.sun.com/products/ejb/docs.html
2. Mark Johnson. A beginner's guide to Enterprise JavaBeans. Java World, October 1998. http://www.javaworld.com/javaworld/jw-10-1998/jw-10-beans.html
3. Nova Laboratories. The Developer's Guide to Understanding Enterprise JavaBeans, 1998. http://www.nova-labs.com
4. Object Management Group. CORBAServices: Common Object Services Specification. 1997. http://www.omg.org/library/csindex.html
5. Sun Microsystems. Java Authentication and Authorization Service, 1999. http://java.sun.com/security/jaas
6. Thomas Riechmann, Franz J. Hauck. Meta Objects for Access Control: Extending Capability-Based Security. In Proc. of New Security Paradigms Workshop, 1997.
7. Jürgen Kleinöder, Michael Golm. Meta Java: An Efficient Run-Time Meta architecture for Java. In Proc. IWOOOS, 1996.