Integrating Security Policies via Container Portable Interceptors

Tom Ritter
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
+49 30 3463 7278
ritter@fokus.fraunhofer.de

Rudolf Schreiner, Ulrich Lang
ObjectSecurity Ltd.
St John's Innovation Centre, Cowley Road,
Cambridge, CB40WS, United Kingdom,
+44 1223 420252
{rudolf.schreiner|ulrich.lang@objectsecurity.com}

ABSTRACT
In the past, it was very common to develop middleware without consideration of security from the very beginning. To integrate security, the middleware that should be protected has to provide appropriate hooks and interfaces, and has to meet the requirements of security. In most cases it is not possible to develop a new, secure middleware from scratch. It is only possible to make minor modification to existing systems. In this paper we describe the successful integration of a CORBA Component based middleware and a policy management framework for the definition, management and enforcement of security policies. Integration is achieved by defining Container Portable Interceptors and QoS Enablers which provide the necessary hooks for interception and provision of context interfaces to integrate the security framework.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Client/Server, Distributed applications; D.4.6 [Security]: Access controls, authentication, information flow controls

General Terms
Management, Security

Keywords
CORBA Portable Interceptors, Policy Management Framework, CORBA Component Model, Container Portable Interceptors

1. INTRODUCTION
Enforcing appropriate security policies in distributed, component based applications is a difficult task. In many cases, the security functionality of standard middleware like CORBA or Enterprise Java Beans is not sufficient to protect an application and its resources. The common solution for this problem still is to directly implement security enforcement in the application source code. While this approach is very flexible, there are several serious issues. First of all, it requires that the persons designing and implementing the application components are security specialists, too. In real life, this is rarely the case. Sometimes the component developers do also not know in which environment the component will be used and are unable to define an appropriate security policy. So it is common that the application programmers simply ignore security. The second issue is the coupling of functional and non functional aspects, which, to a large degree, hinders component reusability. A component can only be reused if the functional and non functional requirements match.

Therefore it is necessary to clearly separate functional and non functional parts of an application. The functional parts, i.e. the business code, are implemented in a component. The non functional parts, for example the enforcing of the security policy, must not be mixed with the business code. It has to be defined by a separate policy and to be enforced by the runtime infrastructure of the component.

While this sounds good in theory, it is hard to do. First of all, a generic framework to define and evaluate security policies is needed. Secondly, it is necessary to integrate the security framework with the middleware platform. The middleware has to provide the necessary hooks to intercept calls and to obtain the information required for the security enforcement.

In the following, we describe the middleware platform we use, the CORBA Component Model (CCM). Then we briefly analyze the CORBA security services and their main issues. We then describe our security framework and its integration into CCM in detail.

1.1 CORBA Components
The CORBA Component Model (CCM) [2] is one of the best platforms for developing large scale distributed systems. It is based on the mature CORBA middleware and adds some more advanced concepts. It also simplifies the usage of some CORBA Services.

CCM enhances the Object Model of CORBA. Figure 1 depicts the features a CORBA Component can have. A component has a component interface (equivalent interface). This interface provides operations for introspections and navigation regarding other components features. A component can provide a set of facets. A facet is a named port providing a specific interface.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RM '05, November 28- December 2, 2005 Grenoble, France Copyright 2005 ACM 1-59593-270-4/05/11... $5.00

Clients of this component call operations on a facet. The facet's counterpart, a receptacle, is a named port where a specific interface can be connected to. A facet of a CORBA Component in server role can be connected to a receptacle of a CORBA Component in a client role. Receptacle ports make dependencies to other interfaces explicit, which helps to minimize wrong configurations and run-time failures by providing type safety.

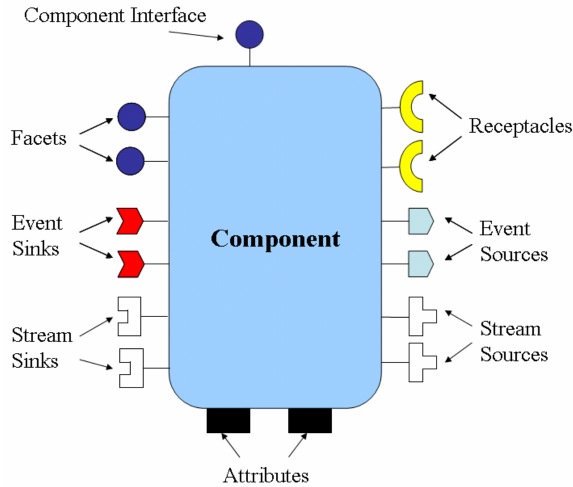

Figure 1. Object Model of CCM

As facets and receptacles are used for operational interactions, the event sources and event sinks are used for event based interactions and message exchange. An event source can publish or emit events of a certain type. Event sinks can consume events of a certain type. A similar port concept for continuous interactions (i.e. data streams) is lately introduced by the OMG to the CORBA Component Model. A stream source port produces streams of data of a specific type while a stream sink port can receive such data. Attributes can be used to configure an instance of a CORBA Component.

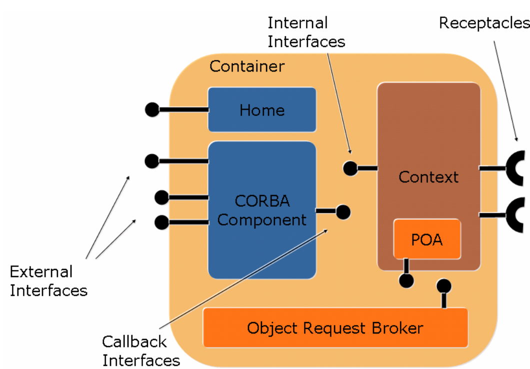

Figure 2. Container of CORBA Component

CCM also defines the container as the run-time environment of a CORBA Component (see figure 2), including an interface to the component which is called context, that provides access to the underlying platform services (e.g. CORBA Services) as well as access to the application environment of the component (e.g. connected interfaces at receptacle ports). Furthermore, the component implementation provides a call-back interface to the container which is in turn used to manage the life-cycle of the component implementation. In the container also resides a Home, which is an implementation of factory and finder pattern for managing component instances.

1.2 CORBA Security

In 1995, the OMG specified the initial version of the CORBA Security Services (CORBASec) specification [4] for CORBA, which includes a security model, as well as interfaces and facilities for applications, administrators, and implementers. The CORBA Security Services specification defines various object interfaces, which provide security functionality like Security Administration, Authentication, Security Context Establishment, Access Control, Communication Protection, Security audit, Non-repudiation.

However, the CORBASec specification is a complex and heavy document (>400 pages) that tries to meet various differing industry requirements in a "one size fits all" fashion. Unfortunately a number of weaknesses have been identified that make this standard fairly unusable in practice.

The most fundamental shortcoming is that CORBA and CORBASec do not provide precise and persistent names for object instances. Instead, it uses the object interface name, which does not precisely describe the object instance (in the case where there are several instances of the same interface). Moreover, due to IDL interface inheritance, it is not possible to reliably obtain the "most derived interface" (i.e. the interface that really contains the called operation) of an operation. This makes a reliable description of the target of an invocation almost impossible; As a result, the CORBASec access control is unusable. An exhaustive discussion on this topic and a workaround which works in some cases can be found in [3] and [6].

A second fundamental shortcoming of CORBASec is the fact that it does not support the technology-neutral representation of security information. On one hand, parts of representation were very strict and seriously limited the description of security attributes. On the other side, other parts are only very loosely defined, making a uniform processing very difficult. In particular, identities (e.g. X.509 or Kerberos) are stored in their native formats. This means that the policy management console and the human administrator need to deal with all the technical peculiarities of the differing underlying technologies. A further problem is that security information can only be interpreted on particular layers of the software stack (e.g. security mechanism, operating system, CORBA POA, application), which further complicates this issue.

In addition, there are a number of other (non-critical) problems that make CORBASec unusable for complex, highly distributed environments:

First, the specification only specifies four access rights (get, set, use, and manage), which only allows 16 different access right combinations. While this makes sense for Unix-style file access, this is totally insufficient for complex object and component invocations. Furthermore, the SECIOP protocol is too heavy and does not deliver the required functionality. Therefore vendors reverted to SSL/TLS as an underlying security mechanism, which was never designed for middleware, does not provide sufficient

functionality to handle required features like delegation, and completely breaks the layering. Also, the management side of CORBASec was never specified. The specification always refers to the (non-existing) "Management Facility" [5] which was not adopted by the OMG.

The security part of the CCM specification is based on the concepts of Enterprise Java Beans security and reuses an underlying implementation of the CORBA security services. In addition to the already described problems in the CCM specification, this also raised the issue of the non matching security models of EJB and CORBASec. EJB security is based on roles, while CORBASec is based on a rights model.

All these issues make CORBASec, even with the proposed workaround for the target identifier, almost unusable for protecting complex distributed applications.

1.3 OpenPMF Policy Management Framework Overview

To overcome the limitations of CORBA security for CCM based applications, we developed the OpenPMF policy management framework [12] to define, manage and enforce security policies in complex distributed systems. While our targeted middleware platform is CORBA/CCM, OpenPMF can be used for protecting other platforms and applications, too.

OpenPMF is based on an abstract model of middleware security, defined in UML. Using this model, security policies can be expressed in terms of attributes describing clients, initiators, targets, operations or other context information. From the abstract model we generated a policy repository to store concrete instantiations of security policies. Policies are defined in a policy description language (PDL) and are fed into the repository using a PDL compiler. During the application startup or policy updates, the security agents, also called Policy Enforcement Points (PEP), in the application obtain the policy from the repository and instantiate it. During runtime, the security agents intercept the invocations and evaluate the policy, based on the invocation's context. If the invocation is not allowed by the policy, the invocation is aborted and an exception is raised. In the following we will describe in detail how context information is described and obtained, because this is one of the critical aspects of middleware security, and how policies are defined and enforced.

Due to space limitations, we will not cover questions of assurance, e.g. how we ensure that the policy is correctly enforced or how the integrity of the security framework itself is protected.

2. CONTEXT INFORMATION

A very important point in adaptive and reflective systems is the description of the context. The context gives information about the invocation or the environment, and is used by the policy evaluator to make its decisions. It consists of information describing both the functional and non-functional aspects of the system. Functional information for example includes the target of the call and the operation to be invoked, it is directly associated with an invocation. Non functional information is in many cases also directly associated with an invocation, for example the client's credentials, in other cases it describes the environment independently of individual invocations, for example time or a position.

One of the biggest issues in the description and processing of context information is the lack of standardization of its format and semantics, mainly because it describes very different information from different sources. It is therefore very hard to define a standard and orthogonal format for the context. If the format is well defined and restrictive, for example a simple string, it can be handled in a standard manner, but it can only express very limited information or its processing is resource consuming, for example if it is used to express time or numbers. If on the other side the format is very flexible, for example a buffer, it can express very different information, but it is difficult to process in a uniform way. Another point to consider here is not only the data itself, but also the description of the type. As already described, this is one of the points where CORBASec failed.

Context information is obtained from different sources, for example the client's credentials are provided by the underlying security mechanism, while the invocation target and operation have to be provided by the middleware platform. This complicates obtaining the context information a lot. Sometimes context information cannot be directly used by the policy evaluator. For example, if the security mechanism supports only the client's identity, but the security policy is described in terms of roles or groups. In this case it is necessary to map the information provided by the security mechanism to the required information, for example by using a directory server. The different sources and formats of context information make its processing in policy management frameworks difficult, because it is impossible to foresee all possibilities in the policy evaluator. In OpenPMF we developed an orthogonal approach for the handling of context information, called Transformers.

The Transformers provide a uniform interface to the underlying data sources. To obtain data from different source, the policy evaluator now can call the same function, whether the data is provided by the middleware, the security mechanisms or by any other source. Transformers can be stacked; the Transformer called by the policy evaluator can obtain data not only directly, for example from a security mechanism, but also from underlying Transformers.

A standard interface to context information is useful, but it does not solve the problem of the information format and semantics. The policy evaluator still does not know how to process this information, for example how to compare the data. We solved this issue by moving the processing to the Transformer itself, because the Transformer knows the data it process. Instead of letting the policy evaluator obtain the information from the Transformer and compare it by itself with a selector from the policy, the policy evaluator just calls the Transformer's compare function with the selector as argument. The context information is obtained automatically by the transformers during the comparison operation.

In addition to the unified handling of context information, the Transformers also allow an abstract definition of security policies. Instead of writing security policies in terms of mechanisms or platform specific attributes like X.509 DNs or Kerberos names, Transformers map these attributes to abstract attributes used in the policy definition.

The Transformer abstracts from all peculiarities of the underlying platforms, the different data sources, data formats and semantics.

This greatly simplifies the policy evaluation and increases the flexibility of OpenPMF.

3. POLICY DEFINITION AND EVALUATION

In most security systems, security policies are defined as access control lists or using policy definition languages like Ponder [17] or SOL [18]. In these systems, the language is a very central aspect of the system.

In our policy management framework, we use a different approach. The central aspect is not the language, but the abstract information model of the policy, the *meta-policy*. This meta-policy gives an abstract way to describe policies, completely independent how the policy is expressed, for example in a file. For the definition of the meta-policy we used the OMG's MetaObject Facilities (MOF) [16]. We defined the meta-policy using an UML model. This meta-model describes how to express policy hierarchies, rules and the entities used for the rule definitions, for example initiators, clients, targets and operations. Meta-policies can be defined in very different ways. For example it is possible to describe specific security models, e.g. Mandatory Access Control. For role based access control (RBAC) this was proposed by Lodderstedt et al. [7]. Such semantically rich meta-policies have the advantage that the policy described can easily be mapped to existing enforcement functionality, if it is based on the same security model. It also gives certain guarantees about the correctness of the policies, because only well formed rules can be stored. On the other side, this approach it is not very flexible. Therefore we decided to define a more flexible repository which is not based on a particular security model. The big advantage (but also the challenge) of this semantically poor model is the possibility to define arbitrary rules, both correct rules and garbage. It is therefore necessary to pay great attention to the policy definition.

From the meta-policy, a policy repository is generated. For this purpose we are using the Medini tool [15] to automatically generate a repository with CORBA IDL interfaces to store and obtain the policy. The mapping of the model to the IDL interfaces is defined by the OMG MOF standard. The meta-policy and the repository derived from it do not solve the problem how to define a policy. For this purpose we developed a simple policy description language (PDL). It is able to describe policy hierarchies and policy rules. In contrast to common trends, we do not use XML for the policy description, because it was the initial intention to manually write the policy. The policy, expressed in a PDL text file, is compiled and loaded into the policy repository. It is also possible, but not yet implemented to use a XACML description of the policy. Vice versa, it is also possible to generate a PDL file from the policy stored in the repository

The policy enforcement is done by *Policy Evaluators*. They are integrated into the middleware's call chain, for example in CORBA they are called by Portable Interceptors and in components by Container Portable Interceptors (COPI). At application startup, the Policy Evaluator obtains the initial security policy from the Policy Repository and instantiates it in an efficient and compact internal representation. It also registers itself at a central Management Daemon. When this is done, the Policy Evaluator is called for all invocations. It iterates through the policy and calls the Transformers associated with the entities in the rules, for example to check the client's identity. When a matching rule is found, the appropriate action is triggered, in most cases this means that the invocation is allowed. If no match is found, the invocation is denied by default, an exception is raised, and a predefined operation can be called.

Policies can be updated at runtime. Keeping the policies consistent is only possible by using a Policy Repository, not directly at the Evaluators. The Evaluators receive an automatic notification whenever the policy they are enforcing is modified. They then reload the policy from the Policy Repository.

Meta-policies, model based and adaptive policy definition, and enforcement are powerful techniques for the development of secure and complex distributed systems. But in practice, this is very hard to implement, if the underlying middleware does not support sufficient interfaces for obtaining context information or the modification of the calls. As already shown, this issue already made access control at the middleware level in CORBA very difficult. It is therefore necessary to analyze already during the development of the middleware how to provide the necessary hooks. In the following we will describe the Container Portable Interceptors (COPI). They are the result of a close cooperation between the container developers and the developers of the adaptive security architecture, which requested many additional features.

4. CONTAINER PORTABLE INTERCEPTORS

As outlined before, it is crucial for evaluating and enforcing security policies to have access to the context, and in particular to have access to the call chain. It is important to have the hooks for putting a security policy in place. The fundamental requirement be able to intercept the call chain (which is not unique to security policy integration) lead to the definition of Portable Interceptors (PI) as part of the CORBA specification [1].

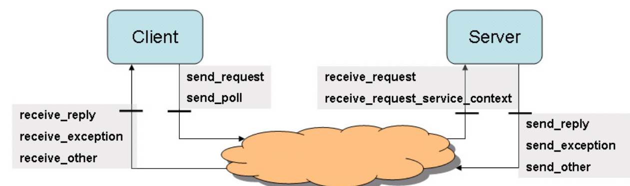

Figure 3. Interception Points

The CORBA specification defines interfaces for the interceptors and for registering them with the ORB. There are a number of interception points defined which are called at specific points in the call chain. Those interception points are also divided into client-side and server-side interception points. Figure 3 gives an overview of the Portable Interceptors.

With the definition of the CORBA Component Model the CORBA Portable Interceptors can not be easily used for hooking user dependent code into the application logic. This is because the PI specification requires the registration of interceptors as part of the initialization process of the ORB. This process, as well as the complete management of the CORBA dependant call chain, is done by the container. Even worse, the ORB creation is done before any user dependent code gets loaded into the container.

Two strategies are available to overcome this issue: Either add proprietary extensions to the container, or adapt the Portable Interceptors to the CORBA Component Model. The authors have decided to follow the second strategy. Based on an initiative of the COACH project [8] the Modelware project [19] works on the definition of the Component Portable Interceptors (COPI). The result is currently in a standardization process and is supported by other companies as well [9]. Furthermore, the Modelware project works on a metamodel and a UML profile for QoS [20] to model QoS properties of component based systems in a generic way and to map such models to CCM platform.

There are two main goals to be achieved with the definition of COPI. The first one is to allow the smooth transformation of CORBA applications based on Portable Interceptors to CORBA Component based application. This means that, similar to the migration from CORBA to CCM, the easy migration from PI to COPI should be possible. The second goal was to extend the concepts of Portable Interceptors to provide more flexibility than the plain CORBA Portable Interceptors give. CORBA Portable Interceptors have one fundamental design feature which prevents their use for very flexible and adaptive system architecture: they are not able to modify the call as such. This means they can only monitor the call, abandon a call or use Forward Location exception to modify the target of a call. Portable Interceptors can not be used to modify operation parameters.

To achieve both goals while not putting to many constraints on container vendors the Container Portable Interceptors are separated into a basic and an extended part. The basic part covers the same functionality as the plain CORBA Portable Interceptors do and the extended part offers additional functionality for modifying requests.

Since basic COPIs are meant for migrating PI code to component level, COPI interception points should be called at the exact same location as PI interception points would be called. This is easily achievable since CCM is designed to be on top of plain CORBA products. This means that for implementing the basic COPIs it is a natural decision to use the PI as foundation. Furthermore, COPIs have method signatures very similar to the ones of the PIs. In contrast, the extended COPIs offer similar but different interception points. While basic interception points are called while the ORB is dispatching a call, the extended interception points are called while the CCM container is dispatching a call. This is schematically depicted in figure 4. However, there is an important difference in the flow-rules between basic and extended interceptors. The extended interceptors can for example return a result for a method call and can prevent the further call processing from reaching the component implementation. This gives the opportunity to modify the behavior of a component by providing different behavior without changing component implementation.

The definition of the COPI interface does not prescribe in any way how to implement the COPI interface and how to register the interfaces within the container. The proposed specification [9] offers a way to do that by using an extended container category and the QoS Enabler concept. This is an optional way of doing it. Vendors can use other approaches to support the usage of COPI interfaces in their containers. The QoS Enabler concept, which is also used for implementing the hooks for integration of security policies, is explained in the next section.

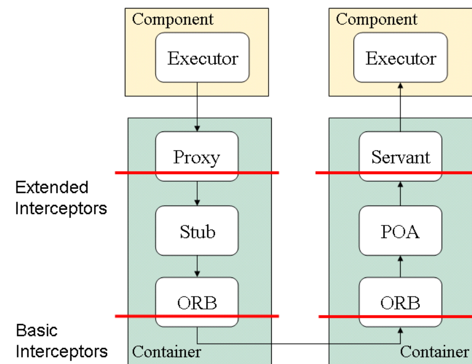

Figure 4. Basic and Extended Interceptors

5. QOS ENABLER

The CORBA Component Model has defined the container model for providing a high level of abstraction to the component implementation. It also offers the possibility to load and to unload user code (components) dynamically by installing or de-installing Homes. Depending on mechanism used by the container vendor and programming language this is realized by loading and unloading shared libraries and it requires sophisticated management of such artifacts at run-time. However, a comprehensive support of this CCM feature is implemented in Qedo [11].

The same principle is applied to the implementation of the COPI interface by defining the QoS Enabler concept. A QoS Enabler is a specialized component that can be loaded into a specialized CCM container and is able to hook in additional functionality. Taking this approach allows usage of plain CCM mechanisms for development and deployment of QoS Enablers (see figure 5). The extension container offers the possibility to register COPIs at the container. Each QoS Enabler is responsible for a specific QoS category. In our case, we used the QoS Enabler concept to implement access control. Here, the QoS Enabler implements the Policy Enforcement Point. During initialization, it loads and instantiates the security policy. At runtime, it intercepts the invocations, calls the Policy Evaluator and, if the invocation is not permitted, raises an exception. QoS Enabler and COPI respectively also provide pointers used by the Transformers for accessing the context information.

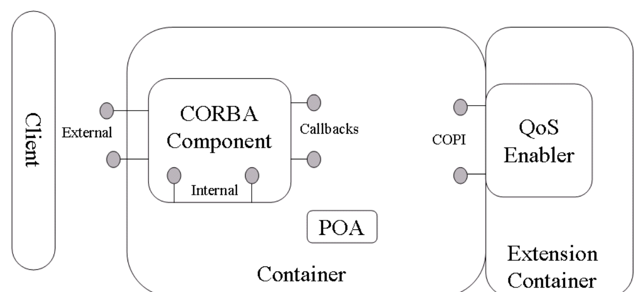

Figure 5. QoS Enabler in extension container

The only condition that has to be ensured at run-time is that on each relevant node a corresponding QoS Enabler is instantiated. The QoS Enabler itself will further be in contact with the policy framework (i.e. the policy repository and OpenPMF management

daemon) to get an update on the security policies that have to be enforced.

At run-time the QoS Enabler at the client side checks the identity of the calling component. This is done at the interception point *send_request.* It adds a security context to the call context. At the server side the QoS Enabler is called at interception point *receive_request*. The QoS Enabler checks the origin identity and the target identity and checks whether there is a policy that has to be applied for that call.

6. APPLICATION IN AIR TRAFFIC CONTROL DOMAIN

A middleware platform based on the concepts described here is currently used for the development of an experimental, secure air traffic control system. It consists of MICO [10] as underlying CORBA implementation, Qedo [11] as extended implementation of the CORBA Components Model and the OpenPMF policy management framework [12]. We also provide an integrated MDA [13] tool chain for the development of the application. The project AD4 [14] is still ongoing, but so far the evaluation of our platform was very successful. Together with an appropriate security infrastructure, for example an IIOP proxy (that is integrated with OpenPMF) to protect against Denial of Service attacks, it is possible to provide the required level of protection. The security enforcement, main access control on invocations and control of information flow, worked as expected, and the central policy management proved to be most useful.

7. CONCLUSION

The separation of functional and non functional aspects is crucial when developing reusable components and secure systems, but is very hard to fully achieve with today's platforms. We presented an integrated secure middleware based on extended CORBA Component Model and the OpenPMF policy management framework. QoS Enablers are used to provide implementations of the Container Portable Interceptors and provides the hooks required for the security enforcement. Policies can now be defined and updated independently of the component, at application startup and at runtime.

Our secure middleware was evaluated in some test applications and is currently used for the development of an experimental air traffic control system. It meets all initial design requirements; we were able to define, manage and enforce appropriate security policies. However, the manual definition of security policies turned out to be cumbersome. This issue will be improved by integrating security into the model based software engineering process, resulting in automated and assisted policy generation. The current snapshot of our system is mainly targeted at relatively static applications, as it does not meet the requirements of highly dynamic systems yet. In the nearer future, we plan to add policy based adaptation.

8. ACKNOWLEDGEMENTS

The presented work partly reflects results of the projects MODELWARE and AD4. MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme. AD4 is a project co-funded by the European Commission under the "Aeronautics and Space" Sixth Framework Programme.

9. REFERENCES

[1] OMG, "Common Object Request Broker Architecture" OMG document number formal/04-03-12

[2] OMG, "CORBA Component Model", OMG document number formal/02-06-65

[3] Lang, U.: Access Policies for Middleware. Ph.D. Thesis, Cambridge University, 2003

[4] Object Management Group: OMG, CORBA Security Services Specification, 2000

[5] Lang, U., Schreiner, R. Developing Secure Distributed Systems with CORBA, ISBN 1-58053-295-0, Artechhouse, 2002

[6] Lang, U., Gollmann, D., and Schreiner, R. Verifiable Identifiers in Middleware Security. *17th Annual Computer Security Applications Conference (ACSAC) Proceedings,* pp. 450-459, IEEE Press, December 2001

[7] Lodderstedt T. et al., Model Driven Security for Process-Oriented Systems, SACMAT 2003, 8th ACM Symposium on Access Control Models and Technologies, 2003, June 2003, Como, Italy, 2003

[8] COACH Consortium. Component Based Open Source Architecture for Distributed Telecom Applications, www.ist-coach.org

[9] OMG,"QoS 4 CCM" revised submission, OMG document umber mars/2005-08-07

[10] MICO, www.mico.org

[11] Qedo, www.qedo.org

[12] ObjectSecurity, OpenPMF project, www.openpmf.org

[13] Object Management Group. Model Driven Architecture Web Page. www.omg.org/mda. Needham, MA, May 2003

[14] AD4 Consortium. 4D Virtual Airspace Management System, http://www.ad4-project.com/

[15] Medini, www.ikv.de/content/ENGLISH/Downloads/medini.htm

[16] OMG, MOF 2.0 final adopted specification, OMG document number ptc/03-10-04

[17] Ponder, www-dse.doc.ic.ak.uk/Research/ policies.ponder.shtml

[18] Bharadwaj, R. SINS, A Middleware for Autonomous Agents and Secure Code Mobility, SEMAS-2002, July 2002, Bologna, Italy

[19] Modelware Consortium. Modelling Solution for Software Systems, www.modelware-ist.org

[20] OMG, UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. OMG document number ptc/050-05-02