PoliCap - Proposal, Development and Evaluation of a Policy Service and Capabilities for CORBA Security

Carla Merkle Westphall*, Joni da Silva Fraga, Michelle Silva Wangham and Lau Cheuk Lung

Universidade Federal de Santa Catarina - LCMI-DAS-UFSC
Campus Universitário - Trindade - Florianópolis - SC - Brazil
PO Box 476 - CEP 88040-900
e-mail: {merkle, fraga, wangham, lau}@lcmi.ufsc.br

Abstract

This paper presents Policap - a Policy Service for distributed applications that use CORBA security model. Policap was proposed for insertion in the JaCoWeb Project context, which is developing an authorization scheme for large-scale networks. This scheme is being developed in order to deal with management of security policies in such networks, simplifying authorization policy implementation. The Policap policy service fills the existing gap in the use of security policies for distributed object programming. The paper further presents the implementation results obtained and an evaluation of these results based on Common Criteria, ISO standard 15408.

Keywords – Security Policies, Security, Authorization Schemes, CORBA.

1 INTRODUCTION

Distributed applications (such as teleconferencing, white-boards, cooperative editing, group decision support systems and banking), which were limited in local networks (e.g.: intranets), are being customized for large-scale networks, such as the Internet, to heed the demands of great corporations. Large-scale networks are characterized by the great space distribution, with an open feature, integrating significant amounts of computational resources designated by its heterogeneity. A recent concern in relation to these new applications is the definition of new services that regard the security requirements in large-scale networks. These services must provide mechanisms that assure the properties of confidentiality, integrity and authenticity, in order to operate correctly, efficaciously, and safety.

Management of security policies in large-scale systems is a great concern today. However, it presents several difficulties, as there is a large number of users, objects and operations, along with the lack of a security policy enforcement and heterogeneous environments, present in large-scale network [1]. Thus, complexity and scaling increase the difficulty of security policy management.

The inherent complexity of distributed object systems cause additional vulnerabilities, which have to be countered by the security architecture. Among these vulnerabilities, the access control on very large systems is problematic, since distributed object systems can scale without limit, and new components are constantly being added, deleted and modified in these environments. In geographically large systems there usually are many different security policy domains that difficult their administration.

Many of these distributed applications follow the object-oriented paradigm and consider CORBA (*Common Object Request Broker Architecture*) the best alternative for adjusting to open systems requirements. As result of this, the OMG introduced, after some years of work, the specifications of the CORBA Security Service (CORBASec) [2], which, if implemented and managed properly, can provide a high level of security for information and applications in large-scale environments. The CORBASec specifications, which still must pass for some extensions (and updates), define a useful set of service interfaces and facilities for implementation of safety applications in heterogeneous distributed systems.

According to the CORBASec specifications, when an object is created in a distributed system, and made visible via CORBA, it automatically becomes a member of one or more security domains. For each security domain, a set of policies is defined, which define a set of rights (permissions) for invocation of the operations

* Doctoral student (CAPES scholarship holder) of Electrical Engineering at UFSC.

on objects of the security domain. However, a detailed analysis of the CORBASec specifications, shows that it still lacks procedures for managing domain members and do not define procedures for including and removing security polices, which are essential for these new applications in large-scale networks [2, 3, 4].

The *PoliCap* policy service, proposed in this paper, aims to provide a policy object management service centralized in a domain of distributed object applications. Our proposals meet the need identified in CORBAsec related to policy object management and were developed in order to act in the model of the *JaCoWeb* project [5].

The *JaCoWeb* authorization scheme (http://www.lcmi.ufsc.br/jacoweb/) now being developed in our laboratories, corresponds to an access control structure based on two levels of control: a global and a local level, in the application object hosts. *PoliCap* establishes the first level of verification, at binding time. The second access control level based on capability mechanisms is carried through in application execution time, reflecting the *PoliCap* policies, allowing verifications on both sides – on the client and server sides – in method invocations. The capability mechanism is set up using abstractions of CORBA itself. This authorization scheme is based on CORBAsec specifications, also integrating aspects of Java and Web security models.

Initially, the paper presents the security model of the CORBA standard in section 2. In section 3, the policy service proposal and the authorization scheme considered are described. Implementation results are shown in section 4 and the prototype evaluation developed is made in section 5 according to the Common Criteria of Security. Section 6 presents some conclusions and related work.

2 CORBAsec – CORBA SECURITY MODEL

The OMG wrote a document defining the main directions for distributed object security [2, 4]. The security model described in this document establishes some procedures involving the authentication and the verification of the authorization in the invocation of a remote method, the security of the communication among objects, in addition to aspects involving procedures for delegating rights, non-repudiation, auditing and security management.

The CORBA security model relates objects and components on four levels of a system: the *application level* with the application objects; the *middleware level* formed by service objects (COSS: *Common Object Services Specification*), ORB services and the ORB core (all on the level of *middleware* CORBA); the *security technology level* composed of the underlying security services; and finally, the basic protection level formed by a combination of operating systems and hardware functionalities. Figure 2 illustrates the levels and main components of CORBAsec in the context of *JaCoWeb* project. The application applet and server represent the application level. ORB services and service objects are built on the ORB core and extend the basic functions with additional qualities or controls, facilitating the distributed object implementation. A combination of ORB services and COSS services is used on the *middleware* level, to implement the CORBAsec. The underlying security technology defines algorithms and protocols used in implementing some CORBAsec service objects.

2.1 CORBAsec service objects

PrincipalAuthenticator, *Credential*, *DomainAccessPolicy*, *RequiredRights*, *AccessDecision*, *SecurityManager*, *PolicyCurrent*, *Current*, *Vault* and *SecurityContext*, introduced in CORBA specifications, constitute the COSS service objects that implement the security controls of a method invocation.

The *PrincipalAuthenticator* object implements the principal authentication service in CORBA with the chief aim of acquiring the principal *credentials*. A credential (*Credential* object) has the privileges of a principal necessary to allow it access to system objects during a session. After acquiring credentials, the principal and objects that act on its behalf as clients can begin the invocations to server objects.

The authorization controls of the model allow the security policies defined to be verified on two levels: on the *middleware* level using service objects during a method invocation and, therefore, transparently to the application; and on the application level, which is aware of the security services of the environment. In this work we restrict ourselves to the transparent authorization controls of security-unaware applications.

In CORBAsec, the security policies are described in the form of *security attributes* of the system resources (*control attributes*) and of the principals (*privilege attributes*). The *DomainAccessPolicy* object represents the access interface to a *discretionary* authorization policy, granting to a set of principals a specified set of rights to perform operations on all objects in the domain [6]. The representation of a particular access policy is defined in table 1.

Privilege Attribute	Delegation State	Rights Granted
Role:bank_manager	Initiator	Corba: gs--
Role:bank_manager	Delegate	Corba: g---
Role:bank_teller	Initiator	Corba: g--u

Table 1. *DomainAccessPolicy* object.

To simplify administration, *DomainAccessPolicy* aggregates principals for access control by using their privilege attributes as subject entries. Some types of grouping are *group* and *role*. There is the possibility of allowing different rights to be granted to a subject depending on whether the principal initiates an invocation (*initiator*) or is participating in a delegation chain of tasks (delegate). Thus, the subject in the traditional sense of an access control matrix is a combination of a privilege attribute and a delegation state [6]. To cope with different needs to express authorization, rights are classified into sets of *rights types* named *rights families*. *CORBAsec* defines only the *Corba* family that contains four types of rights: *g* (*get*), *s* (*set*), *m* (*manage*) and *u* (*use*), although it allows a free definition of other rights families.

An access policy grants rights to privilege attributes according to its delegation state. In contrast, a *RequiredRights* object determines that for the invocation of each operation in the interface of a secure object, some rights are necessary or required (*control attributes*). An example of *RequiredRights* object is shown in Table 2. This table defines the rights required to gain access to each specific operation of an object. There is also a rights combinator that makes it possible to specify whether a principal needs *all* the rights in an operation's required rights entry to execute that operation or whether it is sufficient to match *any* right within the entry. Required rights are assigned to *interfaces* and not to *instances* (all instances of an interface will always have the same required rights).

Required Rights	Rights Combinator	Operation	Interface
Corba:g---	All	See_Balance	Savings_Account
Corba:gs--	Any	Deposit	Savings_Account
Corba:g--u	All	Deposit	Checking_Account

Table 2. *RequiredRights* object for *Savings_Account* and *Checking_Account* interfaces.

All access decisions of object invocations are made through a service object interface known as *AccessDecision*, which determines whether or not an operation to be executed by a specified target object is allowed. The access decisions rely on privilege and control attributes provided by *DomainAccessPolicy* and *RequiredRights* respectively. For example, the policy defined in table 1 grants a principal *bank_teller*, in delegation state *Initiator*, all required rights – *g* and *u* – to execute the operation *Deposit* of the *Checking_Account* interface.

Crucial objects for understanding the dynamic aspects of CORBAsec are the *session objects*. All security service objects (COSS) are accessed through session objects *SecurityManager*, *PolicyCurrent* and *Current* (figure 2), that hold information about the current security context related to the process (*capsule specific*) or to the thread of execution (*thread specific*) respectively [2, 7]. The *SecurityManager* object mantains state information associated with the ORB instance - for example, it keeps references to *RequiredRights* and *AccessDecision* objects, that are used during an invocation. State information associated with policy objects of the current execution context are kept in the *PolicyCurrent* object. The *Current* object, on the server side, keeps the credentials that are sent by the client during the establishment of a association.

The *Vault* and *SecurityContext* objects participate in establishing an secure association. A *secure*

association is defined between client and server objects when there is trust between the entities by means of authentication. Usually, *Vault* object creates credentials on behalf of *PrincipalAuthenticator* object, according to the underlying security technology used in authentication. Moreover, *Vault* object is responsible for creating the security context objects named *SecurityContext*, on both client and server sides in an association, which keeps information about the security context used to protect messages, providing integrity and/or confidentiality.

2.2 Interceptors

In CORBA security model specifications, interceptors implement *ORB services*. Each COSS service related to security is associated with an *interceptor*, whose purpose is to cause the transparent deviation of a method invocation, activating a corresponding service.

In a CORBA security model, two interceptors are designated that act while a method is being requested (figure 2): the *Access Control Interceptor* that on higher level causes a deviation to carry through the access control in the call and, the *Secure Invocation Interceptor* that makes a lower-level interception in order to establish a secure association and provide integrity and confidentiality properties in the corresponding invocation exchanges.

These interceptors defined in CORBAsec are created during the *binding* process between two application objects that are to communicate and are associated with different functionalities at various moments of a method invocation (in both sides).

In *binding time*, the *access control interceptor* is responsible for creating the *AccessDecision* object, updating its reference in the *SecurityManager* object. The *AccessDecision* object, in turn, in *binding time*, is responsible for making domain policy objects (*DomainAccessPolicy*) available. This occurs as a consequence of the *get_domain_policy* operation execution of the domain manager – resulting in the insertion of the *DomainAccessPolicy* reference in *PolicyCurrent* object. The *AccessDecision* also locates the *RequiredRights* object of the environment inserting its reference in the *SecurityManager* object.

In *access decision time*, the *access control interceptor* invokes the *access_allowed* operation of the *AccessDecision* object. The *access_allowed* operation is responsible for allowing or disallowing the invocation, obtaining the rights granted by the *DomainAccessPolicy* object, and comparing these rights with the required rights, obtained from *RequiredRights* object, to execute the operation specified.

The s*ecure invocation interceptor* is responsible, at bind time, for the establishment of secure association between the client and the server. At bind time, this low-level interceptor activates the *Vault* service object in order to create a *SecurityContext* object of the secure association that must be established between the client and the server objects. This *SecurityContext* object provides the security context information and is used, in *message protection time*, by *secure invocation interceptor* to maintain integrity and/or confidentiality.

2.3 Security Technologies

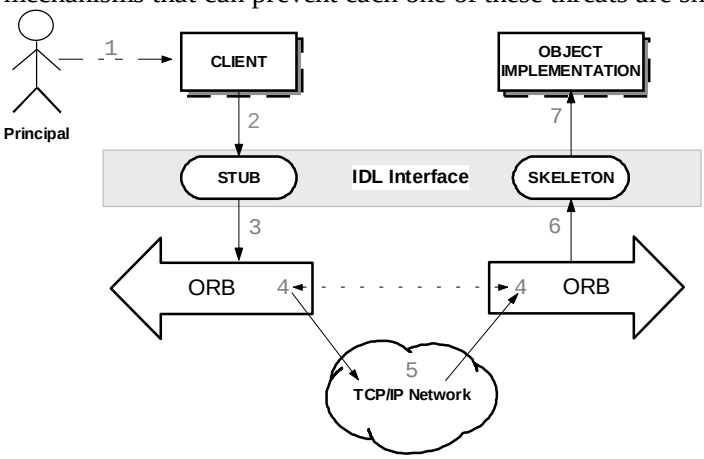
The service objects in the CORBA security model isolate applications and the ORB from the security technology (as in figure 2), which consists of an underlying layer that implement some functionality of related security service objects. The security technology includes services such as authentication, secure association (key distribution, certificates, encryption and decryption), and others. According to the CORBAsec specifications, technologies that may be used to provide these services are [1]: SPKM; Kerberos; CSI-ECMA (based on SESAME and ECMA GSS-API) and SSL (*Secure Socket Layer*)[2].

CORBAsec was initially developed for static applications in restricted environments and cannot be easily adapted to new requirements and trust relationships of Internet-based applications, for example because the code base of CORBAsec is too big, and because firewalls block messages passed between objects. The OMG firewall draft [8] and the integration of SSL into CORBA are first attempts to bring CORBAsec to the Internet [9].

Among the various distributed system technologies available, SSL [10] stands out for being a cryptographic protocol widely used in Internet applications. SSL is an adequate general-purpose protocol for connection-protection in distributed systems, and for assuring authenticity, privacy and integrity in communications through TCP/IP connections. The SSL protocol was chosen as the basic security mechanism for this work, firstly for having been inserted in CORBAsec specification and secondly because it is standard of fact to Internet-based applications.

2.4 CORBA Security Consideration Points

Figure 1 shows the seven points of attacks in the CORBAsec environment [11]. The threats and security mechanisms that can prevent each one of these threats are shown in table 3.



Other threats are:

- Bypass of Security Controls
- Lack or loss of accountability
- Misconfiguration of the system
- Vulnerabilities with no countermeasures

Figure 1. CORBA Security Consideration Points.

Attack Points	Threats	Security Mechanisms
1	• User impersonation • User exceeding assigned authorization	• User Identification • User Authentication and • User Authorization
2	• Undesired use of an object implementation • Request/Response repudiation • Disclosure of data	• Application-layer access control • Non repudiation • Security audit logging • Data protection
3	• Unprotected security-unaware applications • Unwanted revelation of client machine existence	• Client-side object invocation access control • Data protection
4	• Object masquerade • Client masquerade • IOR tampering • Disclosure of request contents • Modification/destruction of request contents	• Authentication between client and object • Encryption between client and object • Delegation controls • Security audit logging
5	• Network eavesdropping • Message tampering • Inability to cross network boundaries (e.g., firewalls)	• Transport encryption • IIOP traversal of firewalls
6	• Unprotected security-unaware applications • Too many object interfaces and implementations to manage individually	• Server-side object invocation access control • Security policy domains
7	• Unauthorized disclosure of specific information to client • Request/response repudiation • Protection of data	• Application-layer access control • Non-repudiation • Security audit logging • Data protection

Table 3. The threats in CORBAsec environment and security mechanisms to prevent them.

3 JaCoWeb Security Framework

The *JaCoWeb* aims to use CORBA security model integrated with Web and Java security models to compose an authorization scheme for distributed applications in large-scale networks. This scheme is being developed in order to carry out the implantation of global policies, representing a great challenge, mainly in large-scale networks, such as the Internet [5].

In this environment, distributed applications are expressed in the form of clients represented by Java applets, and of application server objects, available via CORBA. The *authorization scheme* defines two security control levels: the *global level* and the *local level*. These two levels are actualized in COSS service objects and in security nodes and TCB' s (*Trusted Computing Bases*), respectively. The service objects described in section 2.1, concentrate functions of identification and authentication of users and authorization controls in the access of visible objects on the global level. The security nodes and TCB' s, present in each machine of the system, validate the ways of access to the local resources.

The implementation of an authorization project does not depend only on access control. Other internal controls are also important for the implementation of authorization policies, such as cryptographic controls, authentication and identification services etc. In this work, to implement these cryptographic controls, the SSL was used as the underlying technology. However, the CORBAsec specifications do not present a clear insight (regarding concepts and needs) on the integration of ORB and security technologies without compromising interoperability. In this article, the *JaCoWeb* framework, integrating ORB and SSL, based on the service objects approach, is also presented. It will not alter ORB' s original characteristics and functionality, thus allowing ORB+SSL to perform both conventional and secure connections. Figure 2 shows the main components that implement the *JaCoWeb* framework.

The *PoliCap* is a policy service for distributed objects whose invocations are ruled by the CORBAsec model. *PoliCap* was designed within the context of *JaCoWeb* project and corresponds to a first level of access control verification in the authorization scheme. The second access control level corresponds to a *capability* mechanism. The emphasized boxes are the main contribution of our work, considering the original CORBAsec model and are an evolution of our previous work [5].

The application applet (figure 2), after its authentication, interacts with the CORBA name service (*CosNaming*) [12], to obtain, from the name of the object, the reference or IOR of the server application object. This must allow the binding with the server object. The calls executed by this application applet on a remote application server are subjected to two levels of access control. On the higher level, the verification occurs in *binding time*, and once the requisition is validated, the *PoliCap* policy service provides versions of the *DomainAccessPolicy* policy objects and of the *RequiredRights* object that are used locally in the validation of access requests to the application objects. From this high level verification, capabilities are generated which are validated locally in the remote servers, completing, in this way, the second access control level defined in our scheme.

To construct these two levels of access control, we use the two defined interception levels on CORBA security model and its service objects. The high level interception, on the client side, deviates to the *AccessDecision* service object that obtains, in *binding time*, from the *PoliCap* policy service (detailed in section 3.2), the local objects (*DomainAccessPolicy* and *RequiredRights*) to be used in verifications of *capabilities* mechanism in *access decision time*. On the server side of the application, this high-level interception is used to validate the capability received with the invocation request.

The representation of privileges in the form of capabilities in the authorization scheme is detailed in section 3.3. Besides the controls cited above, the cryptographic controls are also necessary in the scheme and are defined in the CORBA service object form that uses the security technology resident under the ORB (detailed in section 3.1).

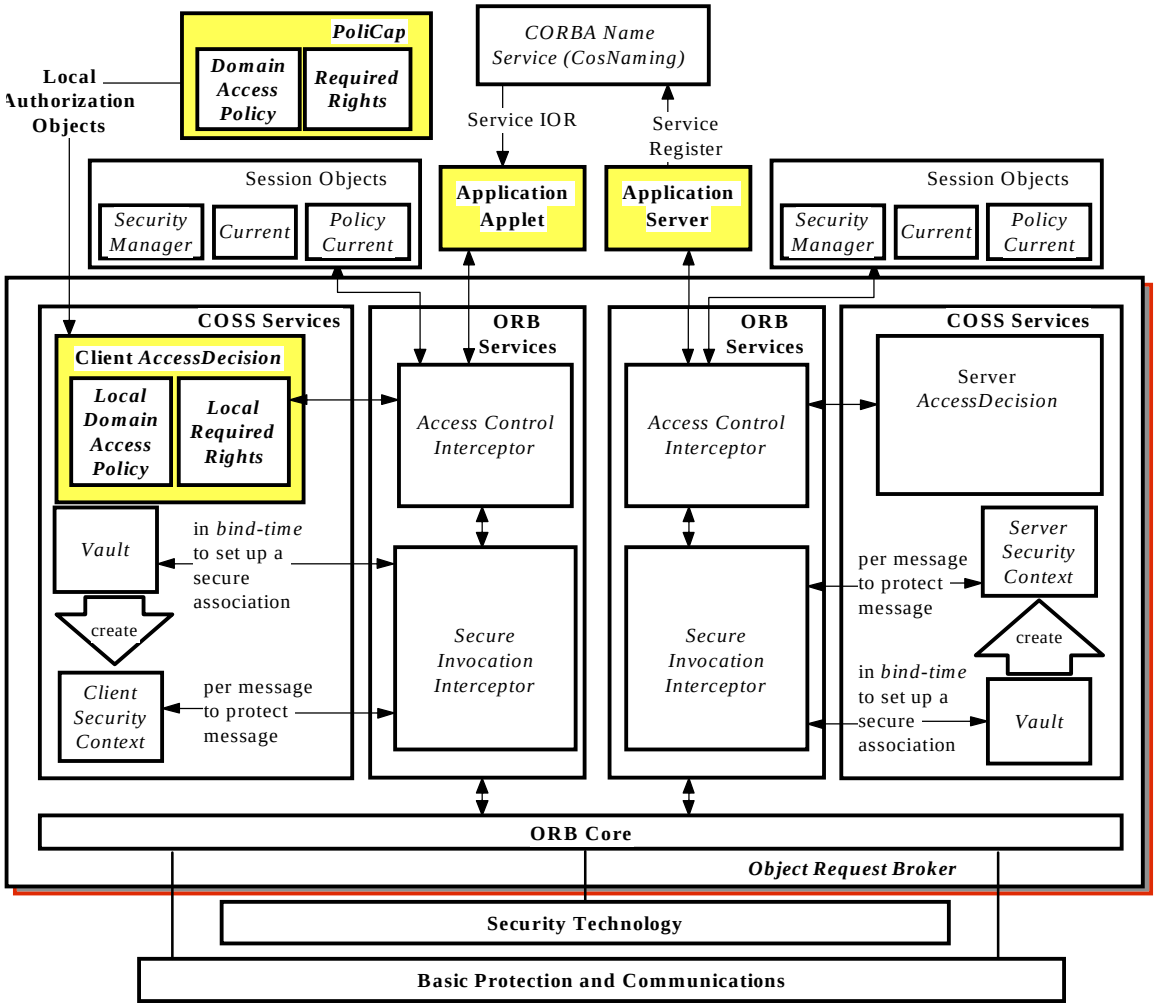


Figure 2. *JaCoWeb* Authorization Scheme.

3.1 ORB +SSL Integration

During the development of CORBAsec, SESAME [13] was the most powerful security mechanism available at that time, therefore the basic concepts of SESAME are found in CORBAsec. Unexpectedly the weaker SSL became more widely used than SESAME. Consequently, CORBAsec on top of SSL is not as powerful as CORBAsec on SESAME, and many features simply no longer match [9].

Two ways of integrating SSL technology and ORB were discussed during the project phase, both according to the CORBAsec specification [2]. In this work, we attempted to define a solution that would allow the ORB to deal with both secure and conventional connections, thus maintaining the ORB' s original functionality. The first alternative consists in implementing the entire SSL specifications as a part of the security service. In this case, the model is completely respected, and security concepts are created and maintained by the security service. In other words, the message interceptor has complete control over the handshake process. The major problem with this implementation is its high cost, since it would be necessary to restructure an available SSL code, in order to have an API to be accessed by the Vault object. The other solution consists in using an SSL package with available executable code, and model it according to the needs. In this case, however, some of the functionality of the message interceptor in controlling the handshake process is lost. The second alternative was chosen, since, not having to implement the entire SSL specifications, the authors could pay special attention to other CORBA security model services.

The framework for integrating ORB/CORBA and SSL support consists in encapsulating an available SSL package in the form of a service object, such as the COSS objects.

3.2 PoliCap

The *PoliCap* policy service, proposed in this work, sets out to provide the central management of policy objects in a domain of distributed object applications, filling in the existing gap of policy object management in CORBAsec specifications. According to these specifications, discretionary security policies are made available through *DomainAccessPolicy* and *RequiredRights* objects. *Administrative applications* are responsible for defining these objects and *operational applications* use them to carry out the access control [2].

According to CORBA security service, when an object is created, it automatically becomes a member of one or more domains (policy domains), being subjected to the security policies of the domain. However, the present specification [2] does not cover procedures either to manage domain members (include, remove, etc) or to manage security policy objects. The security policy domains are formed by a collection of object references that have a set of common security policies and are managed by a *domain manager* object [3]. This manager has the following functions [14]: to provide mechanisms for creating and accessing policy objects of the domain and to keep the domain structures updated. However, interfaces to add new policies (new policy objects) to domains or to modify domain membership are still not standardized. The initial idea to fill in this gap is being proposed in the *Security Domain Membership Management Service* [3], where an extension of the *DomainManager* interface is proposed, with administrative functions that would serve to define and remove policy objects of a domain.

Even with these interfaces not yet standardized in CORBAsec, we felt the need to introduce this policy object management service in the authorization scheme proposed. The service developed – the *PoliCap* – is based on initial documents (*drafts*) released by OMG [3] and surely will not be too far from the specifications that are to be standardized shortly. The *PoliCap* (figure 2) is a service that offers operations, both for *administrative* and *operational* functions concerning policy objects, playing the roles of *domain manager for policy objects* (*DomainAccessPolicy*) and of *rights manager for required rights objects* (*RequiredRights*) in our domain. The authorization scheme proposed (*JaCoWeb*) initially was defined as consisting of a unique domain and the policy service acts as *a central service for managing policies and rights*.

Administrative applications interact with *PoliCap* to manage policies and required rights and, *operational applications* or COSS objects, interact with the policy service to obtain, in *binding time*, policies and required rights necessary for the controls over a method invocation in execution time. The idea is that, in *binding time*, the policy service (domain manager) is to provide the *DomainAccessPolicy* and *RequiredRights* objects that act on an invocation. Figure 3 describes the *PoliCap* interface.

The operation *set_policy* of the *DomainAccessPolicyAdmin interface* associates the authorization policy defined and the corresponding policy object with the policy domain. The operation *delete_policy* of the *DomainAccessPolicyAdmin interface* removes the authorization policy from the domain.

The operation *get_local_domain_policy* of the *DomainAccessPolicyAdmin interface* sets up a *DomainAccessPolicy* object with the effective rights (*GrantedRights*) in the authorization policy related to the privilege attribute and its delegation states (subject). In fact, this operation sets up locally, *in binding time*, a version of the *DomainAccessPolicy*, essential for the validation (*in access decision time*) of several operations present in the same interface. For example, getting the *DomainAccessPolicy* object with the information of table 1 - the global object defined in the domain - to execute the operation: `localDomainAccessPolicy = get_local_domain_policy(SecClientInvocationAccessDiscretionary,'role,authority,bank_teller', initiator)`, the return of this operation is the *DomainAccessPolicy* object – a version shown in table 4 – that has to act locally in the *AccessDecision* object of an invocation.

```
module PoliCap {
#include "SecurityLevel2.idl"
        interface DomainAccessPolicyAdmin:CORBA::DomainManager {
        const CORBA::PolicyType SecClientInvocationAccessDiscretionary = 100;
        void set_policy ( in    CORBA::PolicyType policyType,
                          in    CORBA::Policy    policy);
        void delete_policy (    in    CORBA::PolicyType policyType);
        CORBA::Policy get_local_domain_policy ( in    CORBA::PolicyType    policyType,
                                                in    Security::SecAttribute    priv_attr,
                                                in    Security::DelegationState del_state); };
        interface RequiredRightsAdmin : SecurityLevel2::RequiredRights {
                SecurityLevel2::RequiredRights get_local_required_rights (
                          in    CORBA::Identifier target_interface_name); };
};
```

Figure 3. IDL interface of *PoliCap*.

Privilege Attribute	Delegation State	Granted Rights
Role: bank_teller	Initiator	corba: g--u

Table 4. *DomainAccessPolicy* object returned by the *get_local_domain_policy* operation.

The operation *get_local_required_rights* of the *RequiredRightsAdmin interface* also sets up a local version, with the required rights present in the global *RequiredRights* (centralized) object of the domain policy service. This version of the *RequiredRights* object, set up locally *in binding time*, contains all rows related to an interface, considering that a client object can execute several operations defined in these application object interface, for example, having the *RequiredRights* object (global) in table 2, executing the operation: localRequiredRights = get_local_required_rights (Savings_Account), the returned value of this operation is the *RequiredRights* object (local) shown in table 5.

Required Rights	Rights Combinator	Operation	Interface
Corba:g---	All	See_Balance	Savings_Account
Corba:gs--	Any	Deposit	Savings_Account

Table 5. *RequiredRights* object returned by *get_local_required_rights* operation.

3.3 Capabilities using CORBAsec

In the *CORBAsec* model, access control decisions can be carried out both on the client side and on the server side. The access control in the target object side is defined in CORBAsec specifications as *normal*. But it is also made clear that, when access control verifications are made on the client side, access denials to the server object determine a network traffic decrease. In some ORB's, system integrity considerations can cause the trust in the access control made on the client side alone, to become undesirable [2].

In *JaCoWeb*, the aim is to control the access to the visible distributed objects via CORBA, assuming that, on the first level, the verifications are made by the client object *AccessDecision* in partnership with the *PoliCap* service. The execution of the operations *get_local_domain_policy* and *get_local_required_rights* from the interception of access control in *binding time*, sets up the local objects *DomainAccessPolicy* and *RequiredRights*. These objects are part of the structure necessary in the client object *AccessDecision* to generate *capabilities* to be verified in the server object *AccessDecision*. High-level verification is made only once, during the *binding* of the client and server objects, on the first request. The requests of subsequent operations on the server interface will be validated only by the capability mechanism. The two access control levels reduce the network traffic in the case of access denial, and at the same time the security of the system does not depend exclusively on the client's integrity.

The capability mechanism is not sponsored by *CORBAsec*, although there is the suggestion regarding the

use of *capabilities* in [2], from the abstractions created in the model. It cannot be stated that a universal definition *of the content of* a *capability* exists. Traditionally, a *capability* must contain the reference of an object and a set of rights. *Capability* defines the rights of a holder over the object in question. In [15], a classic *capability* is represented as the triple (*IdObject*, *Rights*, *Random*) containing the name of the object, a set of access rights and a random number, respectively. The number random prevents falsification, normally resulting from a *one-way f* function, applied over the identifier of the object and the rights over that object (*Random = f(IdObject, Rights)*)[1]. When a request arrives on the server side, with its respective *capability*, the *one-way f* function is executed again and its result is checked with the random number sent to detect possible modifications (*tampering*) in the request message. This random number plays the role of a *nonce*, and also assures the '*freshness'* property [17] of the client request.

Just as it is impossible to falsify a *capability*, it must also be impossible re-utilize it or pass it on to third parties. For this reason, normally, in addition to the random number, a client identification field is included in a *capability* for the one who requested the operation. These *capabilities* are protected by ciphering mechanisms when transmitted on the communication support [15].

In the *JaCoWeb* scheme, the *capabilities* are created dynamically at each client request for operations on the server. The *Request* class defined in the ORBA specifications is used in the composition of a *capability* in our scheme. A *capability* in *JaCoWeb* contains in its fields: the *IOR* (*Interoperable Object reference*) *of the server object*, representing the identity of the entity over which a *capability* provides the access rights; *method requested*, representing the right; *identifier of the sender/principal*, representing the identity of the entity requesting the operation; and a *nonce* that assures the '*freshness'* property of the request. In a *Request* for a usual invocation in CORBA, the *IOR* of the server object and the identification of the method requested are already present. The other two *capability* fields, identifier of sender of request and a *nonce* field, must be inserted in *Request* during the high level interception, in the object *AccessDecision* on the client side. The value of *nonce* is calculated as follows: *nonce = f* (identifier of sender, method requested, server IOR, Random); where *f* corresponds to the SHA *one-way hash* function, present in the *java.security* package [18]. The value *Random* is a random number calculated by means of the *java.security.SecureRandom* class of the Cryptographic API of Java JDK 1.2 [18]. This random value is a redundancy that guarantees the *freshness* property of the *Request* message, providing protection against *replay* attacks [15]. The values of *Random* are also inserted into *Request* messages and sent to the server functioning as a message counter (*Random*, *Random+1*, ...). The identifier of the sender (or the one who requested the operation) is obtained from the client credential. After being included to generate the *capability*, the *Request* is represented as in figure 4.

New *Request* with the *capability* (IOR, method, sender, *nonce*)					
Old *Request*			Added fields		
Server IOR	Requested method	...	Identifier of Sender	*Nonce*	*Random*

Figure 4. *Request* structure with added fields to implement the *capability*.

At the *time of access decision*, the access control interceptor *on the server side* verifies the *capability*. In this way, *capabilities* can be built for each of a client's invocations in *CORBAsec*. These *capabilities* are utilized to form a second access control level in the *JaCoWeb* authorization scheme.

4 JaCoWeb IMPLEMENTATION

A prototype – including the *PoliCap* policy service, the *capability* mechanism and SSL as the underlying technology - was developed in our laboratories (figure 5). An application example consisting of a bank system composed of a CORBA server object and a Java client *applet* was constructed to test the implementation of this prototype. The CORBA server object was developed with the tool JacORB 1.0 [12], a *free* Java ORB, and

[1] There are several practical algorithms that implement *one-way hash* functions. The algorithms of MD4, MD5 e SHA are an example of these functions [16].

the client *applet* was implemented with the tool JDK 1.2.1. Netscape *browser* 4.5 was also used as an environment for the interaction between the client and the server. The aim of this implementation was to effect a discretionary policy based on CORBAsec structures [33].

This version of *PoliCap* was concerned with fully developing, first of all, the whole dynamic aspect of the CORBAsec service objects, which is no trivial task, inasmuch as the specification utilized is extremely broad [2] and the dynamism of an application that uses CORBAsec is not described in the literature. For this reason, this initial version of *PoliCap* defines only the local objects *DomainAccessPolicy* and *RequiredRights* defined in a static form *in the binding time* that resides in the client machine. Figure 5 synthesizes the functionalities implemented in the prototype.

Among the objects implemented in this prototype (besides the signed *applet* and application server) are the objects *ClientAccessDecision*, *ServerAccessDecision*, *DomainAccessPolicy*, *RequiredRights*, *SecurityManager* and *PolicyCurrent*. These objects use other CORBA (COSS) services, such as name service. The prototype is limited to a single name domain. The object *PrincipalAuthenticator*, responsible for the authentication and creation of the object *Credential*, was not implemented in this prototype. The prototype credentials are created statically and have security attributes that identify the client rights in the system and the security mechanism that is being utilized (SSL [10], in our case).

The entities that are subject to security controls in our system are the *principals* (with their privilege attributes) and the server application objects (with their control attributes). The experiment implements discretionary policies by verifying access control *on the client side*.

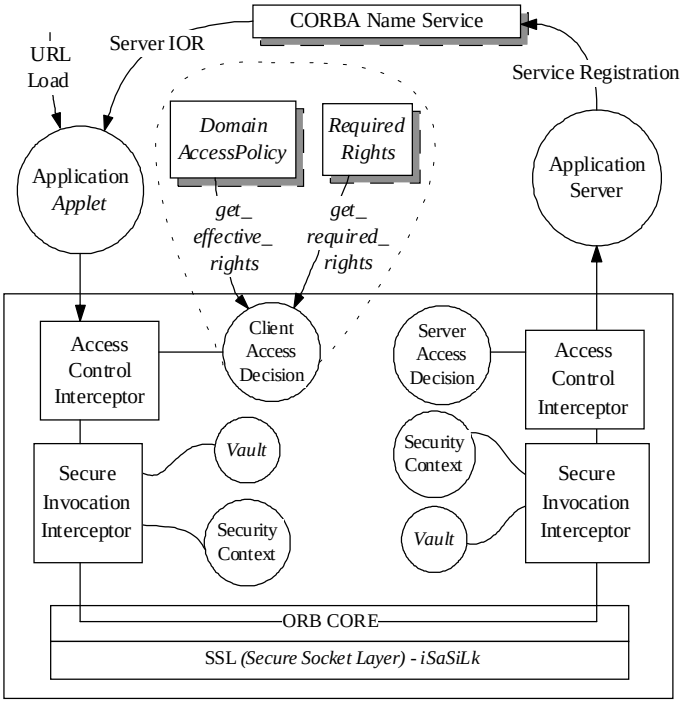


Figure 5. Structure of Prototype Implemented.

The implementation uses as deviation mechanisms the *interceptors* present in JacORB. In the prototype, the access control interceptor in the client machine invokes the object *ClientAccessDecision*, that is responsible for the validation of the access requests for the methods of server object, interacting with the local objects *DomainAccessPolicy* e *RequiredRights*. The method *access_allowed* (figure 6) of object *ClientAccessDecision* obtains the required rights invoking the method *get_required_rights* of the object *RequiredRights* and obtains the granted rights by the *DomainAccessPolicy* invoking the method *get_effective_rights*. It compares the required rights and the granted rights to the privilege attribute to decide whether or not the method to be invoked can be executed.

From this high level verification, the object *ClientAccessDecision* in cooperation with the access control interceptor (that has access to the *Request* CORBA structure) generates *capabilities*, as defined in section 3.3. Using API cryptographic methods of JDK 1.2, in the package IAIK-JCE and used in the prototype for executing ciphering tasks [19], the values of *nonce* and *random* are generated. *The sender identifier* is obtained from the static credential. Once a *capability* has been formed, its fields are inserted in the *Request* CORBA. The object *ServerAccessDecision* verifies capabilities. In the prototype, the package iSaSiLk v.5.2 that implements the SSL v3 in Java [19] was utilized. All negotiation and use of SSL is executed transparently

by iSaSiLk [19]. The iSaSiLk package was chosen because it presented all the functionality required for the proposed framework.

The implementation of objects defined in *PoliCap* will cause our model to take on its original format once again, as shown in figure 2, where local objects are obtained from global objects, thus performing the first global level of the authorization scheme *JaCoWeb*.

```
//   Access_allowed method of the ClientAccessDecision object

public boolean access_allowed(Org.omg.SecurityLevel2.Credentials[] cred_list, Org.omg.CORBA.Object target,
String operation_name, String target_interface_name) {

boolean b = false;
// Obtains required rights – invokes get_required_rights
Org.omg.Security.RightsListHolder rightslist = new Org.omg.Security.RightsListHolder();
Org.omg.Security.RightsCombinatorHolder combinator = new Org.omg.Security.RightsCombinatorHolder();
rights.get_required_rights((Org.omg.CORBA.Object) null, operation_name, "sistemaBancario.idl", rightslist,
combinator);

//extensiblefamily of  JaCoWeb project – SSLCredentials class
Org.omg.Security.ExtensibleFamily extFamily = new Org.omg.Security.ExtensibleFamily((short) 0, (short) 1);
Org.omg.Security.AttributeType[] attributes = new Org.omg.Security.AttributeType[1];
attributes[0] = new Org.omg.Security.AttributeType(extFamily, 5);

// Obtains granted rights of the DomainAccessPolicy object – invokes get_effective_rights
granted = access.get_effective_rights(cred_list[0].get_attributes(attributes), extFamily);
for (int i = 0; i < access.getCount(); i++) {  // Compares required rights with granted rights to grant or deny access
                        if (granted[i] != null) {
                                    if ((granted[i].right).equals(rightslist.value[0].right)) {
                                                i++; b = true; break; }
                        }           }           return b;}
```

Figure 6. *Access_allowed* for the *ClientAccessDecision* object.

5 PROTOTYPE EVALUATION USING SECURITY COMMON CRITERIA

The Common Criteria (***C**ommon **C**riteria – CC*) for the evaluation of security are the result of the effort of various international organizations to develop single security evaluation criteria for systems and products of information technologies *in distributed systems (http://www.commoncriteria.org/)*. These criteria are derived from previous standards, such as TCSEC, ITSEC, CTCPEC and have the collaboration of Canada, France, Germany, Holland, England and the United States. They became standard ISO 15408 in 1999 [20, 21, 22].

The industry, government and academic communities are interested in using these security evaluation criteria (http://niap.nist.gov/cc-scheme/iccc/trackc.html), since there is a growing demand for security assurance of security products and systems. Some products like Microsoft Windows 2000 and Oracle Databases are using CC to evaluate their products. A list of other CC evaluated products and the names of companies involved are described in http://commoncriteria.org/epl/ProductType/all.html. OMG itself [2] suggests using CC as en evaluation tool for distributed objects applications that are based on CORBAsec. OMG is also working with CC in mind, as can be seen at the DOCsec 2000 proceedings (http://cgi.omg.org/meetings/docsec/presentations.html).

The CC [20, 21, 22] define the way to express the security requisites of a system or product, define distinct categories of *functional* and *assurance requirements*. *Functional requirements* define the desired security behavior. *Assurance requirements* are the basis for gaining the confidence that the means of security requested are effectively and correctly implemented. The confidence in the security of an information technology can be obtained through actions performed during the development, evaluation and operation processes.

Some important concepts of this criterion include *TOE, PP* and *ST*. The *object of evaluation* (*TOE – **T**arget **o**f **E**valuation*) is the part of the product or system that is subject to evaluation. In our case, the *TOE* to

be considered is the prototype developed. The *protection profile* (*PP*) is a definition of sets of requisites and goals, regardless of the implementation, that allow the consumers and developers to create standardized sets of security requisites according to their needs. In the case of the evaluation of this prototype, a registered *PP*, designated as *Controlled Access Protection Profile (CAPP)* [24], was utilized and accepted as standard. Security threats to *TOE*, goals, requisites and the specifications for functional and assurance requirements offered by that *TOE* to fulfill the specific requisites, altogether, constitute the main entrances to the ***S**ecurity **T**arget* (*ST*). The *ST* can declare the conformity to one or more of the *PPs* and form *the basis for an evaluation*.

In this way, the CC are used to *develop* PP's and ST's, defining *what a product/system must do* and also, to *evaluate* security features of products/systems against known and understood requirements in *order to gain assurance* that the implementation of the TOE is correct and that the TOE satisfies defined objectives.

The *CC* provides a security guarantee of a *TOE* using the concept of *active investigation*, which is an evaluation of a system or an information technology product in order to *determine its security properties* [21]. Following the example of previous criteria, *CC* also has a set of security guarantee levels named EALs - ***E**valuation **A**ssurance **L**evels*. There are seven levels of guarantee: EAL1 to EAL7.

There are two stages for the evaluation of a TOE: the ST evaluation and the corresponding TOE evaluation. To this end, an *ST* known as *JaCoWeb-ST*, was developed for the prototype and some of its components are here described. The evaluated[2] *ST* operates in conformity with [24] reaching the level EAL3 of *CC*.

EAL3 provides a moderate level of assurance, meaning that the prototype is methodically tested and checked. The security functions are analyzed using a functional specification, guidance documentation, and the high-level design of the TOE to understand the security behavior. The analysis is supported by independent testing of a subset of the TOE security functions, evidence of developer testing based on the functional specification and the high level design, selective confirmation of the developer test results, analysis of strengths of the functions, and evidence of a developer search for obvious vulnerabilities (e.g. those in the public domain). Further assurance is gained through the use of development environment controls, TOE configuration management, and evidence of secure delivery procedures [23].

5.1 JaCoWeb-ST

The *JaCoWeb-ST*, as defined for an *ST*, is composed of a description of the *TOE*, by the threats to which the *TOE* is subject, by security policies, security goals that determine the functional and assurance requisites that must exist in the system and a declaration of conformity to some *PP* available and certified (*CAPP*).

The description of *TOE* has already been presented in section 4. The *threats* that the prototype can expect to prevent are designated as follows: *T.ACCESS*, where a user obtains access to information or resources of the system without full authorization, *T.CAPTURE*, where an invader can modify or obtain information by listening on the transmission line, *T.INTEGRITY*, where the integrity of information transmitted can be affected due to user or transmission errors, *T.SECRET*, where a user of the prototype, either intentionally or accidentally, can obtain confidential information from the system without permission and *T.IMPERSON*, where an invader can obtain access to information or resources by impersonating an authorized user of the prototype. Some kinds of threats cannot be contained in the prototype: there is no way of holding any user responsible for his/her acts, since there are no auditing services.

Among the existing types of security policies in the prototype is the discretionary policy designated - *P.DAC* that seeks to prevent the above-mentioned threats from taking place. The rights of access to resources provided by the prototype are available through the objects *DomainAccessPolicy* and *RequiredRights*.

---

[2] *JaCoWeb* Security Group conducted an informal evaluation of the ST against the ASE (Security Target Evaluation class) requirements presented in part 3 of the CC. Although a working draft, the ST was deemed to be in a reasonable state to allow the evaluation to proceed. NSA (http://www.radium.ncsc.mil/tpep) is responsible for formally evaluating the ST.

The security goals are determined so as to establish necessary functional requisites for containing the threats defined for a system and to enforce the established security policy. Among the security goals defined that are in accordance with *CAPP* are the following: *O.AUTHORIZATION*, which establishes that the security functions of the prototype must be implemented so that only authorized users may have access to the system, *O.DISCRETIONARY_ACCESS*, which establishes that the security functions of the prototype must provide permission for access to the resources and information based on the attributes of subject privilege and *O.MANAGE*, which establishes that the prototype must support the authorized administrator of the system.

Each security goal is implemented by a set of functional and assurance requisites defined for the prototype. As the prototype is being evaluated in conformity with [24], the same set of functional requisites and of assurance requisites can be assumed. *Functional requirements* define aspects such as audit, user data protection, identification and authentication, security management and protection of TOE security functions of the *JaCoWeb* prototype. *Assurance requirements*, in turn, define the following aspects: configuration management aspects, delivery and operation components, development specifications that describe the implementation, guidance documents necessary for correct administration and use of the prototype, life cycle support (which defines physical, procedural, personnel, and other security measures that are necessary to protect the confidentiality and integrity of the TOE design and implementation in its development environment), security testing (that involves depth, coverage, functional and independent tests results) and vulnerability assessment.

As an example, the goal *O.DISCRETIONARY_ACCESS* is implemented by the following security requisites: FDP_ACC.1 (discretionary access control policy, implemented by the objects of CORBAsec), FPD_ACF.1 (access control function implemented by the object *ClientAccessDecision*), FIA_ATD.1 (definition of user attributes made statically in the prototype), FIA_USB.1 (link between user and principal established in case the user is *manager* or *client* of the bank server of the prototype), FMT_MSA.1 (management of object security attributes, implemented by the object *RequiredRights* of CORBAsec) and FMT_MSA.3 (initialization of static attributes, implemented to manage the object policy in the prototype).

A *summary specification* of this ST describes the security features of the *JaCoWeb* prototype. This prototype has cryptographic support provided by *iSaSiLk Toolkit* [19], user data protection performed by discretionary security policy objects of the CORBA security model, identification and authentication of the mobile code (*applet* Java) through signed *applets*, security management performed by the security administrator of the system and security functions protection by enforcing security in the ORB level.

5.2 JaCoWeb *Evaluation Report Results*

The TOE evaluation was conducted by following the evaluator actions elements defined by the EAL3 requirements [23] using the evaluated *JaCoWeb-ST* as the basis. As a result, a *Final Evaluation Report* was written in order to document the prototype evaluation. Performing each one of the evaluator action elements, the *JaCoWeb Security* team, in conformity with all other requirements defined in [24], concluded that the prototype is considered a *TOE* level EAL3, meaning the prototype is methodically tested and checked.

Evaluation evidences performed during the *JaCoWeb* evaluation related to the tests results and to the vulnerability analysis, some of the key points of the CC evaluation methodology [25, 26], are described here and are also available in Portuguese in the Internet links references.

5.2.1 Tests results

The purpose of this activity is to determine whether the TOE behaves as specified in the design documentation and in accordance with the TOE security functional requirements specified in the ST. This is accomplished by determining that the developer has tested the security functions against its functional specification and high-level design, gaining confidence in those test results by performing a sample of the developer's tests, and by independently testing a subset of the security functions. The tests activity at EAL3 contains sub-activities related to the following components: ATE_COV.2 (evaluation of coverage),

ATE_DPT.1 (evaluation of depth), ATE_FUN.1 (evaluation of functional tests) and ATE_IND.2 (evaluation of independent testing) [23].

Testing is a dynamic method for verification and validation, where the system to be tested is actually executed and the behavior of the system is observed [27]. Different levels of testing are often employed. *Unit testing* is used for testing a module or a small number of modules. Its objective is to detect coding errors in modules. During *integration testing*, modules are combined into sub-systems, which are then tested. The goal here is to test the system design. In *system testing* and *acceptance testing*, the entire system is tested. The goal here is to test the system against the requirements, and to test the requirements themselves.

There are two approaches to testing, *functional* and *structural*. In functional testing, the internal logic of the system under testing is not considered and the *testcases* are decided from the specifications or the requirements. It is often called "black box testing". Equivalence class partitioning, boundary value analysis and cause-effect graphing are examples of methods for selecting *testcases* for functional testing [27]. Structural testing is concerned with testing the implementation of the *software*. The *testcases* are decided entirely on the internal logic of the program/module under testing. Although a structural criterion could be specified, the procedure for selecting *testcases* is left to the tester. It is often called "coverage testing". The most common structural criteria are statement coverage (execution of all system statements) and branch coverage (execution of logic branches of the system).

Having proper *testcases* is central to successful testing. Although all the faults in a program cannot be practically revealed by testing and, due to economic limitations, the goal of *testcase* selection is to select *testcases* such that the maximum possible number of faults is detected by the minimum possible *testcases*. *Testcases* selection is still not a simple mechanical process. Knowledge and creativity of the tester are still important, despite the availability of tools that select *testcases* to ensure coverage.

Testing usually starts with the definition of a test plan, which is the basic document guiding the entire testing of the software. Is specifies the levels of testing and the units to be tested. For testing different units, the *testcases* are specified and then executed.

In order to obtain tests results for the *JaCoWeb Security* prototype, a test plan was defined (http://www.lcmi.ufsc.br/~merkle/TestPlan.html). Two types of testing were performed: *unit* tests and *system* tests. The units tested were the application objects (application *applet*, bank server and name server) and CORBAsec service objects (*AccessDecision*, *RequiredRights*, *DomainAccessPolicy* and *Current*). *Testcases* are available at http://www.lcmi.ufsc.br/~merkle/TestCases.html.

Unit testing was based on branch coverage and the objective was to cover 95% of the prototype logic branches. Unit testing was applied for each CORBAsec service object listed above. An UML state diagram was made for each one of them. A unit test report is available at http://www.lcmi.ufsc.br/~merkle/UnitTesting.html.

System testing used valid and invalid set of values (equivalence class partitioning), limit boundary values, special sets of values and cause-effect graphing to define *testcases*. System testing was applied for the application applet, the bank server and the CORBAsec service objects dealing with access control and secure invocation tasks. UML *usecases* diagrams were designed to describe application objects and UML class diagrams were used to describe CORBAsec service objects functionalities. A system test report is available at http://www.lcmi.ufsc.br/~merkle/SystemTesting.html.

5.2.2 Vulnerability Analysis

The vulnerability analysis of security applications is one of the key points of the CC evaluation methodology [23]. The purpose of the vulnerability assessment activity is to determine the existence and exploitability of flaws or weaknesses in the TOE in the intended environment. This determination is based upon analysis performed by the developer and the evaluator, and is supported by evaluator testing. Vulnerability analysis is necessary, considering the needs of the current market that demands larger trust in the security of the products [28].

The vulnerability assessment activity at EAL3 contains sub-activities related to the following components [23]: AVA_MSU.1 (evaluation of misuse); AVA_SOF.1 (evaluation of strength of TOE security functions) and AVA_VLA.1 (evaluation of vulnerability analysis). Here we focus on the AVA_VLA.1 item of the vulnerability analysis.

In order to find obvious vulnerabilities, *penetration tests* was used [29]. It consists in the use of hacking tools to attack the systems. The use of those tools was made by members of the *JaCoWeb* Security project, with the purpose of verifying the vulnerabilities of the *JaCoWeb* system.

The configuration used in the vulnerabilities analysis consists of three computers Intel P133 32Mb of a local network, executing the *JaCoWebSecurity* package Version 1.0, the Netscape Communicator 4.5 browser (only in the client computer), Microsoft Windows 95 and FreeBSD 2.2.8 operating systems, the Apache Web Server 1.3.9, the client application *applet*, the application server object and CORBA name server. Figure 7 shows the used configuration.

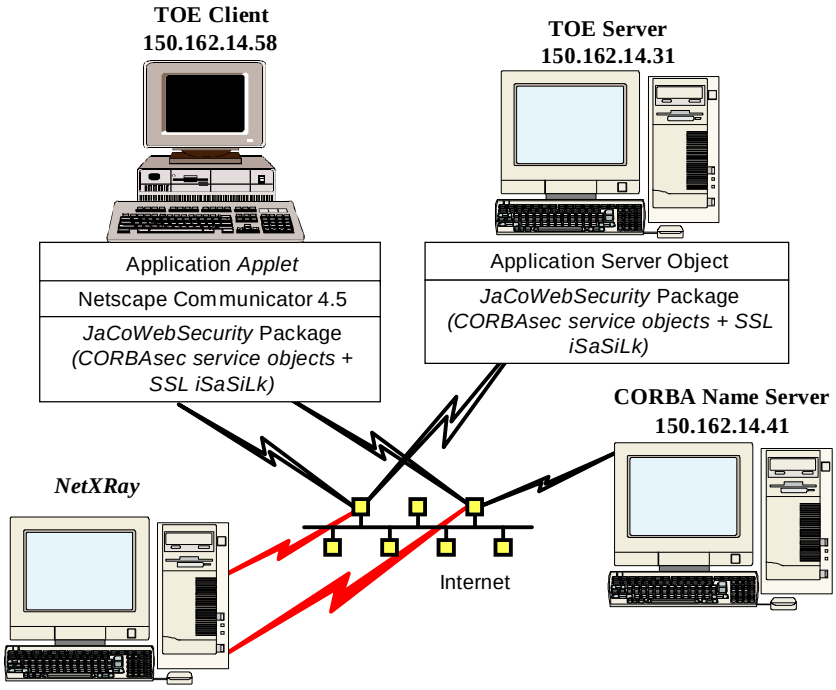


Figure 7. Configuration used in the Vulnerability Analysis and Penetration Tests.

For each of the attack points considered in the CORBAsec model (Table 3), the *JaCoWeb* project provides at least one security mechanism to prevent the occurrence of those attacks (Table 5). We chose for accomplishing the penetration tests using analysis tools (*sniffers*), available in the Internet, to verify if point of attack number five (the communication net), can disclose any sensitive information. *NetXRay* was used for the penetration tests.

NetXRay is a *basic sniffer* that is able monitor the network traffic, capture of packages and the generation of spurious packages to be inserted in the network. The demonstration version was used, and for being demonstrative, it has limitations in the number of packages that can be captured, and also, in the buffer available to store captured data. This tool won the prize of Product of the Year of 1999 performed by the *Network Magazine* and it was the tool of analysts' choice, during research accomplished by*PC Week Magazine*. *NetXRay* is mentioned as one of the tools of contemporary hacking to accomplish penetration tests, in SANS Institute security symposium in 1999 [29]. More information on the product can be obtained in http://www.sniffer.com/.

During penetration tests, the three computers showed in Figure 7 were used. The computer with

150.162.14.58 IP address executed the client application *applet*. The 150.162.14.41 IP address computer executed the CORBA name server and the 150.162.14.31 IP address computer run the bank server.

Attack Point	CORBAsec Security Mechanisms	*JaCoWeb* Security Mechanisms
1	• User Identification • User Authentication • User Authorization	• User Identification • User Authentication • User Authorization
2	• Application-Layer Access Control • Non-Repudiation • Security Audit Logging • Data Protection	• ---------- • ---------- • ---------- • Data protection
3	• Client-side Object Invocation • Access Control • Data Protection	• Client-side Object Invocation • Access Control • Data Protection
4	• Authentication Between Client • and Object • Encryption Between Client • and Object • Delegation Controls • Security Audit Logging	• Authentication Between Client • and Object (via SSL) • Encryption Between Client • and Object (via SSL) • ---------- • ----------
5	• Transport Encryption • IIOP Traversal of Firewalls	• Transport Encryption • ----------
6	• Server-side Object Invocation Access • Control • Security Policy Domains	• Server-side Object Invocation Access • Control • ----------
7	• Application-Layer Access Control • Non-Repudiation • Security Audit Logging • Data Protection	• ---------- • ---------- • ---------- • Data Protection

Table 5 – *JaCoWeb* Security Mechanisms.

For the capture of the packages exchanged between client applet (150.162.14.58) and name server (150.162.14.41), it is possible to observe in the Figure 8, in the fields of data, that the operation requested to the name server can be discovered (*resolve*), and also, what is the name of the server that will be accessed (*banco service*). That is possible because to establish the *first* communication with the name server, the information is exchanged in clear. The next communication exchanges use SSL for ciphering, as well as all communication performed between the client *applet* (150.162.14.58) and the bank server (150.162.14.31), preventing them from disclosing any type of information.

In Figure 8, it can be seen that the network ports used during communication are revealed easily by the tool. Having that information, the possibility of using other kind of tool that sends spurious packages for those ports, causing in that way the service denial, cannot be discarded.

Using the *NetXRay* tool, we can make the following considerations about *JaCoWeb* Vulnerabilities:

- There is the possibility of an intruder obtaining and or modifying the method to be executed by the CORBA name server, in the client' s first request (this is a *CosNaming* problem and concern);
- The denial of service is impossible of being avoided;
- The audit services implementation would increase security level in the points of attacks 2 and 7 (Figure 1);
- The implementation of security policies domains would increase security level in the point of attack 6 (Figure 1).

In that way we verified some vulnerabilities, not considered obvious, that could be explored for the

accomplishment of an attack in *JaCoWeb*. In spite of that, we can affirm that those vulnerabilities do not represent a great risk for the system.

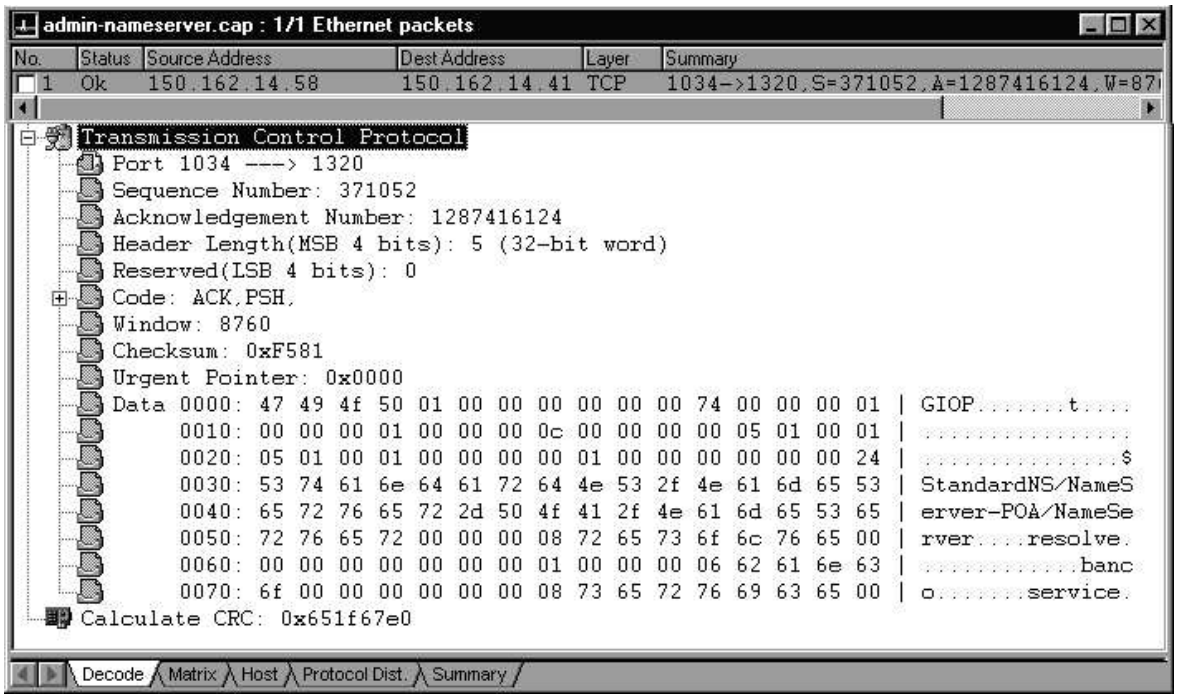


Figure 8. Data captured during Application *Applet* and CORBA Name Server communication.

6 CONCLUSIONS

The literature presents some studies on security policy management in large-scale environments. CORBAsec domain management service specification itself [3] is not standardized.

The access control architecture of the project *Cherubim* [30] was developed using the concept of *capabilities*, that carry the access rights of the *principal* to the server machine where the authorization process takes place. The project does not utilize the CO*RBAsec COSS* objects, implementing its own policy objects instead. The ORB utilized is *JacORB* [12] and the cryptographic services are implemented with *Java IAIK* cryptographic API [19].

Control [7] is an ORB that extends the *ORBAsec* with the *CORBAsec COSS* services that implement authentication services, secure transmission of messages and automatic access control. The discretionary policy is established with an access control language used for server objects. The interception of access control is executed on the server object side. Security policy management is carried out using the domain management module.

Comparing *PoliCap* with the experiences presented, we can verify that *PoliCap* is a service that performs access control *on the client side*, unlike the proposals described. *PoliCap* combines characteristics of *Cherubim*, such as that of *capabilities* and of *Control*, with the additional feature of restricted use of standardized objects or objects still undergoing standardization in CORBAsec.

PoliCap fills in an existing gap in security policy management in the model CORBAsec, actualizing the first access control level of the project *JaCoWeb*. The second access control level is developed with the use of the *capabilities* proposed for the CORBAsec model. *PoliCap* and the *capabilities* for CORBAsec provide an important contribution to the managing of authorization policies in large-scale networks. The two access control levels reduce the network traffic in the case of access denial, and at the same time the security of the system does not depend exclusively on the client's integrity. The description of the dynamic functioning relating to service objects and CORBAsec interceptors is also important.

The prototype is a true experiment with the implementation of the CORBAsec discretionary model, considering that there are few experiments of this type available. This prototype facilitates management and administration of security policies, enabling security to be guaranteed on the ORB level. The security guaranteed on the ORB level has advantages, such as security transparency, for applications and greater confidence in the security services that are always executed. Non-discretionary or mandatory policies are future aims in our prototype.

In large-scale networks the authorization process necessarily goes through the connection of a variety of name servers, each one responsible for a specific domain of objects and users. In each name domain, the controls of the authorization scheme must be present, centered in service objects pertinent to the considered domain. In other words, each name domain must have its *Policy Service* and *PrincipalAuthenticator* objects centering the global controls on persistent objects and users of the domain. The X.500 specifications provide means for these connections among different contexts of names based on *alias* mechanisms [31]. The *alias* can be understood as a local designation in a domain, identifying an object or user as non-local, and it allows the search in resolving names to be extended to other domains. Initially, the authorization scheme covers a single domain, but to make feasible its use in large-scale networks, the possible use of LDAP (*Lightweight Directory Access Protocol*) - an implementation of directory services based on the X.500 and sufficiently used nowadays [32] - is envisioned.

This prototype provided subsidies for the evaluation concerning standard ISO 15408, and we conclude that it meets the level EAL3. Results related to the vulnerability analysis, obtained during the *JaCoWeb* security evaluation, point out some vulnerabilities that could be taken advantage of for a possible attack. For example, the possibility to know which communication port is used among the application objects. These results are important in the context of the evaluation of the *JaCoWeb* project, because they supply data on the vulnerabilities of security mechanisms implemented in the project. We considered that *JaCoWeb* does not possess significant *obvious vulnerabilities* that could be explored for the accomplishment of an attack.

The experience on using *Common Criteria* in a security evaluation process deserves some opinions about. We observe that the higher level of security implies the higher cost of an evaluation. There are many documents about CC, and too many details to be covered. Despite that, CC provides a complete and broad framework for security evaluation and it is an excellent guideline for designing, implementing and evaluating security features of a security technology system. Using CC was a good experience for the *JaCoWeb Security* team.

Future perspectives that can be mentioned are the implementation of audit services and of security policies domains, as defined by *PoliCap*, to improve the security of the points of attack 2, 7 and 6 as well as the use of another types of tools for the accomplishment of penetration tests to identify other existing vulnerabilities.

REFERENCES

[1] Bob Blakley, "The Emperor' s Old Armor," In *Proc. of the 1996 ACM NSPW*, 1996, pp. 2-16, ACM.

[2] OMG, "Security Service:v1.5," *OMG Document Number 00-06-25*, June 2000. (ftp://ftp.omg.org/pub/docs/formal/00-06-25.pdf).

[3] OMG, "Security Domain Membership Management Service," *Doc. orbos/99-07-21*, Aug. 1999.

[4] Bob Blakley, "CORBA Security: An Introduction to Safe Computing with Objects", *The Addison-Wesley Object Technology Series*, 1999.

[5] Carla M. Westphall and Joni S. Fraga, "A Large-scale System Authorization Scheme Proposal Integrating Java, CORBA and Web Security Models and a Discretionary Prototype," *IEEE LANOMS' 99*, pp. 14-25, Rio de Janeiro - Brazil, 1999.

[6] G. Karjoth, "Authorization in CORBA Security," In *Proceedings of the Fifth ESORICS*, Lecture Notes in Computer Science, pp. 143-158, Springer-Verlag, Berlin Germany, September 1998.

[7] Adiron Inc., "Control – Access Control for ORBAsec SL2 V 1.0 Alpha," *Adiron Center*, Syracuse University, Dec. 1999.

[8] OMG, "Joint Revised Submission CORBA/Firewall Security," *Doc. orbos/98-05-04*, Jun. 1998.

[9] A. Alireza , U. Lang , M. Padelis, R. Schreiner and M. Schumacher, "The Challenges of CORBA Security," In: *Workshop Sicherheit in Mediendaten*, June 2000, to appear, published by Springer.

[10] A. Freier, P. Karlton and P. C. Kocher, "Secure Socket Layer 3.0," *Internet Draft*, Nov. 1996.

[11] D. Chizmadia, "An Introduction to the Security Specifications of the Object Management Group," ppt slides. In: *Proceedings of the Fourth DOCsec 2000 - Distributed Object Computing Security Workshop*, April 2000, U.S.A (http://www.omg.org/meetings/docsec/).

[12] Gerald Brose, "JacORB – A free Java ORB," *Freie Universität Berlin*, Institut für Informatik, Berlin, 1999 (http://www.inf.fu-berlin.de/~brose/jacorb/).

[13] Joris Claessens, "A Secure European System for Applications in a Multi-vendor Environment," https://www.cosic.esat.kuleuven.ac.be/sesame/, 2000.

[14] OMG, "ORB Interface," *OMG Document 99-07-08*, June 1999.

[15] Li Gong, "A Secure Identity-Based Capability Systems," *In Proc. of the 1989 IEEE Symposium on Security and Privacy*, pp. 56-63, Oakland, California, May 1989.

[16] W. Stallings, "Cryptography and Network Security:Principles and Practice," *Prentice Hall*, 2nd edition, July 1998.

[17] Martin Abadi and Roger Needham, "Prudent Engineering Practice for Cryptographic Protocols," *IEEE Transactions on Software Engineering*, Vol. 22, Number 1, pp. 6-15, 1996.

[18] Sun M. Inc., "Java Cryptography Architecture API Specification & Reference," Oct. 1998.

[19] Graz - University of Technology, "iSaSiLk 2.5 User Manual," *Inst. for Applied Information Processing and Communications*, Graz University of Technology, Nov. 1999 (http://jcewww.iaik.at/iSaSiLk/isasilk.htm).

[20] ISO/IEC, "Common Criteria for Information Technology Security Evaluation," *In Part 1:* Introduction and general model, ISO/IEC 15408-1, December 1999.

[21] ISO/IEC, "Common Criteria for Information Technology Security Evaluation," *In Part 2: Security Funcional Requirements*, ISO/IEC 15408-2, December 1999.

[22] ISO/IEC, "Common Criteria for Information Technology Security Evaluation," *In Part 3: Security Assurance Requirements*, ISO/IEC 15408-2, December 1999.

[23] CC Project Sponsoring Organisations, "Common Methodology for Information Technology Security Evaluation,", *In Part 2: Evaluation Methodology*, August 1999.

[24] Information Security Systems Organization, "Controlled Access Protection Profile," National Security Agency, Oct. 1999. (http://www.radium.ncsc.mil/tpep/library/protection_profiles/index.html).

[25] K. Jamer - CSE Canada, "Common Evaluation Methodology Special Topic: Testing," ppt slides. In: *ICCC First International Common Criteria*, 23-25 May 2000, Baltimore, land Mary U.S.A. (http://niap.nist.gov/cc-scheme/iccc/trackd.html)

[26] J. Straw, "Common Evaluation Methodology Special Topic: Vulnerability Analysis," ppt slides. In: *Proceedings of the ICCC First International Common Criteria*, May 2000, Baltimore, Maryland, U.S.A (http://niap.nist.gov/cc-scheme/iccc/trackd.html).

[27] Pankaj Jalote, "An Integrated Approach to Software Engineering," *Springer-Verlag New York Inc.*, ISBN 3-540-97561-6, 1991.

[28] A. K. Ghosh et al., "An Automated Approach for Identifying Potential Vulnerabilities in Software," In: *Proceedings of the 1998 IEEE Symposium on Security and Privacy* , May 1998, Oakland, U.S.A, pp. 104-114.

[29] T. J. Klevinsky, "Contemporary Hacking Tools and Their Use in Penetration Testing," Course. In: *FCSC99 - The Federal Computer Security Conference*. Course Day, May 1999, Baltimore, MD, U.S.A (http://www.cio.org/sans99/).

[30] Campbell, Roy and Qian Tin, "Dynamic Agent-Based Security Architecture for Mobile Computers," *Proc. of the Second* PDCN '98, Australia, December 1998.

[31] ITU-T. "Autentication Framework," *ITU-T Recommendation X.509*, Nov. 1993.

[32] Johner, H., et al. "Understanding LDAP," SG24-4986-00 (http://www.reedbooks.ibm.com).

[33] M. Wangham, "Study and Implementation of a Discretionary Authorization Scheme based on CORBAsec Specification," CPGEEL-DAS-UFSC, Master Thesis, Florianópolis, Brazil, March 2000.