Security Services Specification

Version 1.5, May 2000

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, “Need for Object Services.”

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG’s vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.
- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.

- *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Service Design Principles

Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as "building blocks."

Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

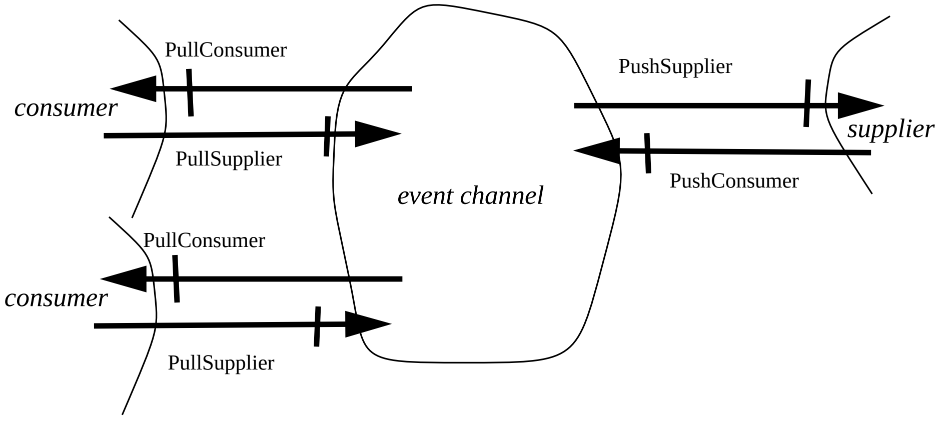
Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer, PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. This is shown graphically in the figure below.



An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

The graphical notation shown in the above figure is used throughout the service specifications. An arrow with a vertical bar is used to show that the target object supports the interface named below the arrow and that clients holding an object reference to it of this type can invoke operations. In shorthand, one says that the object reference (held by the client) supports the interface. The arrow points from the client to the target (server) object.

A blob (misshapen circle) delineates a conspiracy of one or more objects. In other words, it corresponds to a conceptual object that may be composed of one or more CORBA objects that together provide some coordinated service to potentially multiple clients making requests using different object references.

Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service.
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms.

Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

Interface Style Consistency

Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

Explicit Versus Implicit Operations

Operations are always explicit rather than implied (e.g., by a flag passed as a parameter value to some “umbrella” operation). In other words, there is always a distinct operation corresponding to each distinct function of a service.

Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by “normal” clients.

Acknowledgments

The following companies submitted and/or supported parts of the *Security Service* specification:

- AT&T Global Information Solutions Company
- Digital Equipment Corporation
- Expersoft Corporation
- Groupe Bull
- Hewlett-Packard Company
- IBM (in collaboration with Taligent, Inc.)
- International Computers Limited
- Netscape Communications Corporation
- Northern Telecom LImited
- Novell, Inc.
- Siemens Nixdorf Informationssysteme AG
- SunSoft, Inc.
- Tandem Computer Inc. (in collaboration with Odyssey Research Assoc., Inc.)
- Tivoli Systems, Inc.
- Visigenic Software, Inc.

Service Description 1

This specification incorporates material that was adopted in three separate specifications related to security:

- CORBA Security Rev 1.1 (formal/97-12-22)
- Common Secure Interoperability 1.0 (orbos/96-06-20)
- CORBAsecurity/SSL Interoperability (orbos/97-02-04)

All these documents are therefore superseded by this chapter.

Associated with this document, are documents ptc/98-01-03, and ptc/98-01-04, which contain associated changes to the CORBA Core that have been recommended jointly by the Security RTF and the Core RTF. Also associated with this document are the outputs of the C++ and Java language mapping RTFs that had co-terminus delivery dates with the Security 1.2 RTF.

Contents

This chapter contains the following topics, separated into sections.

1.1 Introduction to Security

1.1.1 Why Security?

Enterprises are increasingly dependent on their information systems to support their business activities. Compromise of these systems either in terms of loss or inaccuracy of information or competitors gaining access to it can be extremely costly to the enterprise.

Security breaches, which compromise information systems, are becoming more frequent and varied. These may often be due to accidental misuse of the system, such as users accidentally gaining unauthorized access to information. Commercial as well as government systems may also be subject to malicious attacks (for example, to gain access to sensitive information).

Distributed systems are more vulnerable to security breaches than the more traditional systems, as there are more places where the system can be attacked. Therefore, security is needed in CORBA systems, which takes account of their inherent distributed nature.

1.1.2 What Is Security?

Security protects an information system from unauthorized attempts to access information or interfere with its operation. It is concerned with:

- **Confidentiality**. Information is disclosed only to users authorized to access it.
- **Integrity**. Information is modified only by users who have the right to do so, and only in authorized ways. It is transferred only between intended users and in intended ways.
- **Accountability**. Users are accountable for their security-relevant actions. A particular case of this is non-repudiation, where responsibility for an action cannot be denied.
- **Availability**. Use of the system cannot be maliciously denied to authorized users.

Availability is often the responsibility of other OMA components such as archive/ restore services, or of underlying network or operating systems services. Therefore, this specification does not address all availability requirements.

Security is enforced using security functionality as described below. In addition, there are constraints on how the system is constructed. For example, to ensure adequate separation of objects so that they don't interfere with each other and separation of users' duties so that the damage an individual user can do is limited.

Security is pervasive, affecting many components of a system, including some that are not directly security related. Also, specialist components, such as an authentication service, provide services that are specific to security.

The assets of an enterprise need to be protected against perceived threats. The amount of protection the enterprise is prepared to pay for depends on the value of the assets, and the threats that need to be countered. The security policy needed to protect against these threats may also depend on the environment and how vulnerable the assets are in this environment. This document specifies a security architecture which can support a variety of security policies to meet different needs.

1.1.3 Threats in a Distributed Object System

The CORBA security specification is designed to allow implementations to provide protection against the following:

- An authorized user of the system gaining access to information that should be hidden from him.
- A user masquerading as someone else, and so obtaining access to whatever that user is authorized to do, so that actions are being attributed to the wrong person. In a distributed system, a user may delegate his rights to other objects, so they can act on his behalf. This adds the threat of rights being delegated too widely, again causing a threat of unauthorized access.
- Security controls being bypassed.
- Eavesdropping on a communication line, so gaining access to confidential data.
- Tampering with communication between objects - modifying, inserting, and deleting items.
- Lack of accountability due, for example, to inadequate identification of users.

Note that some of this protection is dependent on the CORBA security implementation being constructed in the right way according to assurance criteria (as specified in Appendix D, "Guidelines for a Trustworthy System") and using security mechanisms with the right characteristics. Conformance to the CORBA security interfaces is not enough to ensure that this protection is provided, just as conformance to the transactional interfaces (for example) is not enough to guarantee transactional semantics.

This specification does not attempt to counter all threats to a distributed system. For example, it does not include facilities to counter breaches caused by analyzing the traffic between machines.

More information about security threats and countermeasures is given in Appendix E, "Guidelines for a Trustworthy System".

1.1.4 Summary of Key Security Features

The security functionality defined by this specification comprises:

- **Identification** and **authentication** of principals (human users and objects which need to operate under their own rights) to verify they are who they claim to be.

- **Authorization** and **access control** - deciding whether a principal can access an object, normally using the identity and/or other privilege attributes of the principal (such as role, groups, security clearance) and the control attributes of the target object (stating which principals, or principals with which attributes) can access it.
- **Security auditing** to make users accountable for their security related actions. It is normally the human user who should be accountable. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.
- **Security of communication** between objects, which is often over insecure lower layer communications. This requires trust to be established between the client and target, which may require **authentication of clients to targets** and **authentication of targets to clients**. It also requires **integrity protection** and (optionally) **confidentiality protection** of messages in transit between objects.
- **Non-repudiation** provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.
- **Administration** of security information (for example, security policy) is also needed.

This visible security functionality uses other security functionality such as **cryptography**, which is used in support of many of the other functions but is not visible outside the Security services. No direct use of cryptography by application objects is proposed in this specification, nor are any cryptographic interfaces defined.

1.1.5 Goals

The security architecture and facilities described in this document were designed with the following goals in mind. Not all implementations conforming to this specification will meet all these goals.

1.1.5.1 Simplicity

The model should be simple to understand and administer. This means it should have few concepts and few objects.

1.1.5.2 Consistency

It should be possible to provide consistent security across the distributed object system and associated legacy systems. This includes:

- Support of consistent policies for determining who should be able to access what sort of information within a security domain that includes heterogeneous systems.
- Fitting with existing permission mechanisms.
- Fitting with existing environments, for example, the ability to provide end-to-end security even when using communication services, which are inherently insecure.

- Fitting with existing logons (so extra logons are not needed) and with existing user databases (to reduce the user administration burden).

1.1.5.3 Scalability

It should be possible to provide security for a range of systems from small, local systems to large intra- and inter-enterprise ones. For larger systems, it should be possible to:

- Base access controls on the privilege attributes of users such as roles or groups (rather than individual identities) to reduce administrative costs.
- Have a number of security domains, which enforce different security policy details but support interworking between them subject to policy. (This specification includes architecture, but not interfaces for such interdomain working.)
- Manage the distribution of cryptographic keys across large networks securely and without undue administrative overheads.

1.1.5.4 Usability for End Users

Security should be available as transparently as possible, based on sensible, configurable defaults.

Users should need to log on to the distributed system only once to access object systems and other IT services.

1.1.5.5 Usability for Administrators

The model should be simple to understand and administer and should provide a single system image. It should not be necessary for an administrator to specify controls for individual objects or individual users of an object (except where security policy demands this).

The system should provide good flexibility and fine granularity.

1.1.5.6 Usability for Implementors

Application developers must not need to be aware of security for their applications to be protected. However, a developer who understands security should be able to protect application specific actions.

1.1.5.7 Flexibility of Security Policy

The security policy required varies from enterprise to enterprise, so choices of security features should be allowed. An enterprise should need to pay only for the level of protection it requires, reducing the level (and therefore costs) for less sensitive information or when the system is less vulnerable to threats. The enterprise should be able to balance the costs of providing security, including the resources required to implement, administer and run the system, against the perceived potential losses incurred as the result of security breaches.

Particular types of flexibility required include:

- **Choice of access control policy**. The interfaces defined here allows for a choice of mechanisms, ACLs using a range of privilege attributes such as identities, roles, groups, or labels. Details are hidden except from some administrative functions and security aware applications that want to choose their own mechanisms.
- **Choice of audit policy**. The event types which are to be audited is configurable. This makes it possible to control the size of the audit trail, and therefore the resources required to store and manage it.
- Support for **security functionality profiles** as defined either in national or international government criteria such as TCSEC (the US Trusted Computer Evaluation Security Criteria) and ITSEC (the European Information Technology Security Evaluation Criteria), or by more commercial groups such as X/Open, is required.

1.1.5.8 Independence of Security Technology

The CORBA security model should be security technology neutral. For example, interfaces specified for security of client-target object invocations should hide the security mechanisms used from both the application objects and ORB (except for some security administrative functions). It should be possible to use either symmetric or asymmetric key technology.

It should be possible to implement CORBA security on a wide variety of existing systems, reusing the security mechanisms and protocols native to those systems. For example, the system should not require introduction of new cryptosystems, access control repositories, or user registries. If the system is installed in an environment that also includes a procedural security regime, the composite system should not require dual administration of the user or authorization policy information.

1.1.5.9 Application Portability

An application object should not need to be aware of security, so it can be ported to environments that enforce different security policies and use different security mechanisms. If an object enforces security itself, interfaces to Security services should hide the particular security mechanisms used (e.g., for authentication). The application security policy (for example, to control access to its own functions and state) should be consistent with the system security policy. For example, use should be made of the same attributes for access control. Portability of applications enforcing their own security depends on such attributes being available.

1.1.5.10 Interoperability

The security architecture should allow interoperability between objects including:

- Providing consistent security across a heterogeneous system where different vendors may supply different ORBs.
- Interoperating between secure systems and those without security.
- Interoperating between domains of a distributed system where different domains may support different security policies, for example, different access control attributes.
- Interoperating across systems that support different security technology.

This specification includes an architecture that covers all of these, at least in outline, but does not give specific interfaces and protocols for the last two. Interoperability between domains is expected to have limited functionality in initial implementations, and interoperability between security mechanisms is not expected to be supported.

1.1.5.11 Performance

Security should not impose an unacceptable performance overhead, particularly for normal commercial levels of security, although a greater performance overhead may occur as higher levels of security are implemented.

1.1.5.12 Object Orientation

The specification should be object-oriented:

- The security interfaces should be purely object-oriented.
- The model should use encapsulation to promote system integrity and to hide the complexity of security mechanisms under simple interfaces.
- The model should allow polymorphic implementations of its objects based on different underlying mechanisms.

1.1.5.13 Specific Security Goals

In addition to the security requirements listed above, there are more specific requirements that need to be met in some systems, so the architecture must take into account:

- **Regulatory requirements**. The security model must conform to national government regulations on the use of security mechanisms (cryptography, for example). There are several types of controls, for example, controls on what can be exported and controls on deployment and use such as limitations on encryption for confidentiality. Details vary from country to country; examples of requirements to satisfy a number of these are:
  - Allowing use of different cryptographic algorithms.
  - Keeping the amount of information encrypted for confidentiality to a minimum.
  - Using identities for auditing which are anonymous, except to the auditor.
- **Evaluation criteria for assurance**. The security functionality and architecture must allow implementations to conform to standard security evaluation criteria such as TCSEC, ITSEC, or Common Criteria (CC)[1]for security functionality and assurance

(which gives the required level of confidence in the correctness and effectiveness of the security functionality). It should allow assurance and security functionality classes or profiles up to about the E3/B2 level. However, the specification also allows systems with lower levels of security, where other requirements such as performance are more important.

1.1.5.14 Security Architecture Goals

The security architecture should confine key security functionality to a trusted core, which enforces the essential part of the security policy such as:

- Ensuring that object invocations are protected as required by the security policy.
- Requiring access control and auditing to be performed on object invocation.
- Preventing (groups of) application objects from interfering with each other or gaining unauthorized access to each other's state.

It must be possible to implement this trusted computing base so it cannot be bypassed, and kept small to reduce the amount of code which needs to be trusted and evaluated in more secure systems. This trusted core is distributed, so it must be possible for different domains to have different levels of trust.

It should also be possible to construct systems where particular Security services can be replaced by ones using different security mechanisms, or supporting different security policies without changing the application objects or ORB when using them (unless these objects have chosen to do this in a mechanism or policy-specific way).

The security architecture should be compatible with standard distributed security frameworks such as those of POSIX and X/Open.

1.2 Introduction to the Specification

1.2.1 Document Overview

This document specifies how to provide security in stand-alone and distributed CORBA-compliant systems. Introducing Object Security services does not in itself provide security in an object environment; security is pervasive, so introducing it has implications on the Object Request Broker and on most Object services, Common Facilities and object implementations.

This document defines the core security facilities and interfaces required to ensure a reasonable level of security of a CORBA-compliant system as a whole. The specification includes:

[1] Version 1 or 2.

- A security model and architecture which describe the security concepts and framework, the security objects needed to implement them, and how this counters security threats.
- The security facilities available to applications. This includes security provided automatically by the system, protecting all applications, even those unaware of security. The security facilities can also be used by security-aware applications through OMG IDL interfaces defined in this specification.
- The security facilities and interfaces available for performing essential security administration.
- The security facilities and interfaces available to ORB implementors, to be used in the production of secure ORBs.
- A description of how Security services affect the CORBA 2 ORB interoperability protocols.
- A description of different levels of secure interoperability that are possible.
- A description of how these levels of interoperability can be provided using a select set of popular security mechanisms and protocols.

Items not included in this specification are:

- Support for interoperability between ORBs using different security mechanisms, though interoperability of different ORBs using the same security mechanism is supported.
- Audit analysis tools, though an audit service that both the system and applications can use to record events is included.
- Management interfaces other than essential security policy management interfaces, as management services are beyond the scope of this chapter. The security policy management interfaces were viewed as a necessary feature of this specification as it is not possible to deploy a secure system without defining and managing its policy.
- Interfaces to allow applications to access cryptographic functions for use, for example, in protecting their stored data. These interfaces are not provided for two reasons: first, cryptography is generally a low-level primitive, used by Security Service implementors but not needed by the majority of application developers; and second, providing a cryptographic interface would require addressing a variety of difficult regulatory and import/export issues.
- Specific security policy profiles.

The security model and architecture specified is extensible, to allow addition of further security facilities later.

1.2.1.1 Normative and Non-normative Material

This specification contains normative and non-normative (explanatory) material. Only sections Section 2.3, “Application Developer’s Interfaces,” on page 2-71 through Section 3.8, “DCE-CIOP with Security,” on page 3-105 and Appendices B and D are normative.

1.2.2 CORBA Security and Secure Interoperability Feature Packages

CORBA security and Secure Interoperability is structured into several feature packages which are enumerated below. These are used to structure the specification as well as to specify the conformance requirements.

- **Main Security Functionality Packages**. There are two packages:
  - *Level 1*: This provides a first level of security for applications which are unaware of security and for those having limited requirements to enforce their own security in terms of access controls and auditing.
  - *Level 2*: This provides more security facilities, and allows applications to control the security provided at object invocation. It also includes administration of security policy, allowing applications administering policy to be portable.

  An ORB must provide at least one of these packages before it can claim to be a Secure ORB. For a definitive conformance requirement see Appendix C, “Conformance Details.”

- **Optional Security Functionality Packages**. These provide functions that are expected to be required in several ORBs, so are worth including in this specification, but are not generally required enough to form part of one of the main security functionality packages specified above. There is only one such option in the specification.
  - *Non-repudiation*: This provides generation and checking of evidence so that actions cannot be repudiated.

- **Security Replaceability Packages**. These packages specify if the ORB is structured in a way that allows incorporation of different Security services, and if so how they can be incorporated. There are two possibilities:

  1. ***ORB Services replaceability package***: The ORB uses interceptor interfaces to call on object services, including security ones. It must use the specified interceptor interfaces and call the interceptors in the specified order. An ORB conforming to this does not include any significant security specific code, as that is in the interceptors.

  2. ***Security Service replaceability package***: The ORB may or may not use interceptors, but all calls on Security services are made via the replaceability interfaces specified in Section 2.5, “Implementor’s Security Interfaces,” on page 2-143. These interfaces are positioned so that the Security services do not need to understand how the ORB works (for example, how the required policy objects are located), so they can be replaced independently of that knowledge.

  An ORB can provide Security by directly implementing the Security feature package 1 or 2 into it without making use of any of the facilities provided by the Replaceability feature packages. But in that case, the standard security policies defined in this specification cannot be replaced by others, nor can the implementation of the Security services be replaced. For example, it would not be possible to replace the standard access policy by a label-based policy if at least one of the replaceability packages is not supported. Note that some replaceability of the security mechanism used for security associations may still be provided if the implementation uses some standard generic interface for Security services such as GSS-API[11].

An ORB that supports one or both of these replaceability packages together with a couple of basic ORB operations as discussed in Appendix D, “Conformance Details” is said to be *Security Ready*[2]. Such an ORB does not in itself support any security functionality but is ready to host security functionality that is implemented to use the facilities of the Security Replaceability package to hook Security into it.

- **Common Secure Interoperability (CSI) Feature packages**: These feature packages each provide different levels of secure interoperability. There are three functionality levels for Common Secure Interoperability (CSI). All levels can be used in distributed secure CORBA compliant object systems where clients and objects may run on different ORBs and different operating systems. At all levels, security functionality supported during an object request includes (mutual) authentication between client and target and protection of messages - for integrity, and when using an appropriate cryptographic profile, also for confidentiality.

  An ORB conforming to CSI level 2 can support all the security functionality described in the CORBA Security specification. Facilities are more restricted at levels 0 and 1. The three levels are:

  1. ***Identity based policies without delegation (CSI level 0)***: At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target, and this cannot be delegated to further objects. If further objects are called, the identity will be that of the intermediate object, not the initiator of the chain of object calls.

  2. ***Identity based policies with unrestricted delegation (CSI level 1)***: At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target. The identity can be delegated to other objects on further object invocations, and there are no restrictions on its delegation, so intermediate objects can impersonate the user. (This is the impersonation form of simple delegation defined in Section 2.1.6, “Delegation,” on page 2-13.)

  3. ***Identity & privilege based policies with controlled delegation (CSI level 2)***: At this level, attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions, so the initiating principal can control their use. Optionally, composite delegation is supported, so the attributes of more than one principal can be transmitted. Therefore, it provides interoperability for ORBs conforming to all CORBA Security functionality.

  An ORB that interoperates securely must provide at least one of the CSI packages. For the definitive statement on conformance requirements see Appendix D.

[2] While this may sound strange, it is still true that a Secure ORB need not be a Security Ready ORB.

- **SECIOP Interoperability package**. An ORB with the SECIOP Interoperability package can generate and use security information in the IOR and can send and receive secure requests to/from other ORBs using the GIOP/IIOP protocol with the security (SECIOP) enhancements defined in Section 3.2, “Secure Inter-ORB Protocol (SECIOP),” on page 3-34 (if necessary), if they both use the same underlying security technology.
- **Security Mechanism packages**: The choice of mechanisms and protocol to use depends on the mechanism type required and the facilities required by the range of applications expected to use it. This specification defines how the following four security protocols can be used as the medium for secure interoperability under CORBA:

  1. ***SPKM Protocol***: This protocol supports identity based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on the definition in [20]. The use of SPKM in CORBA interoperability is based on the SECIOP extensions to IIOP.

  2. ***GSS Kerberos Protocol***: This protocol supports identity based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0). The GSS Kerberos protocol is based on [12] which itself is a profile of [13]. The use of Kerberos in CORBA interoperability is based on the SECIOP extensions to IIOP.

  3. ***CSI-ECMA Protocol***: This protocol supports identity and privilege based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0). For keys assigned to principals, it has two options:
     - It can use either secret or public key technology.
     - It uses public key technology for keys assigned to trusted authorities.

     The CSI-ECMA protocol is based on the ECMA GSS-API Mechanism as defined in ECMA 235, but is a significant subset of this - the SESAME profile as defined in [16]. It is designed to allow the addition of new mechanism options in the future; some of these are already defined in ECMA 235. The use of CSI-ECMA in CORBA interoperability requires the SECIOP extensions to IIOP.

  4. ***SSL protocol***: This protocol supports identity based policies without delegation (CSI level 0). The SSL protocol is based on the definition in [21]. The use of SSL in CORBA interoperability does not depend on the SECIOP extensions to IIOP.

- **SECIOP Plus DCE-CIOP Interoperability**: An ORB with the Standard plus DCE-CIOP secure interoperability package supports all functionality required by standard secure interoperability package, and also provides secure interoperability (using the DCE Security services) using the DCE-CIOP protocol.

  An ORB that interoperates securely must do so using one of these protocol packages. For the definitive statement on conformance requirements see Appendix D.

The requirements that must be satisfied by a conformant ORB are enumerated in Appendix D. The conformance statement required for a CORBA conformant security implementation is defined in Appendix D. This includes a table that can be filled to show what the ORB conforms to.

1.2.3 Feature Packages and Modules

The IDL specified in this chapter is partitioned into modules that closely reflect the feature packaging scheme described above. The Security module holds definitions of common data structures and constants that most other modules depend on. The relationship is as shown in Table 1-1.

Table 1-1 Feature Packages and Modules

Feature Package	Primary Module	Also Depends on
Security Functionality Level 1	SecurityLevel1	Security CORBA, TimeBase
Security Functionality Level 2	SecurityLevel2	Security, CORBA, TimeBase SecurityLevel1 SecurityAdmin
Non Repudiation	NRservice	Security, SecurityLevel2 CORBA, TimeBase
Security Service Replaceability	SecurityReplaceable	Security, CORBA, TimeBase SecurityLevel2
ORB Service Replaceability	Interceptor	CORBA
CSI Level 0, 1 and 2	SECIOP	CORBA
SECIOP	SECIOP	Security, CORBA, TimeBase, IOP
SPKM, Kerberos, CSI-ECMA	SECIOP	Security, CORBA, TimeBase, IOP
SSL	SSL	Security, CORBA, TimeBase, IOP
DCE-CIOP	DCE_CIOPSecurity	Security, CORBA, TimeBase, IOP

The specification is based on a general three layer architecture as shown in Figure 1-1 on page 1-14, with the interfaces defined in each module positioned as shown in the figure.

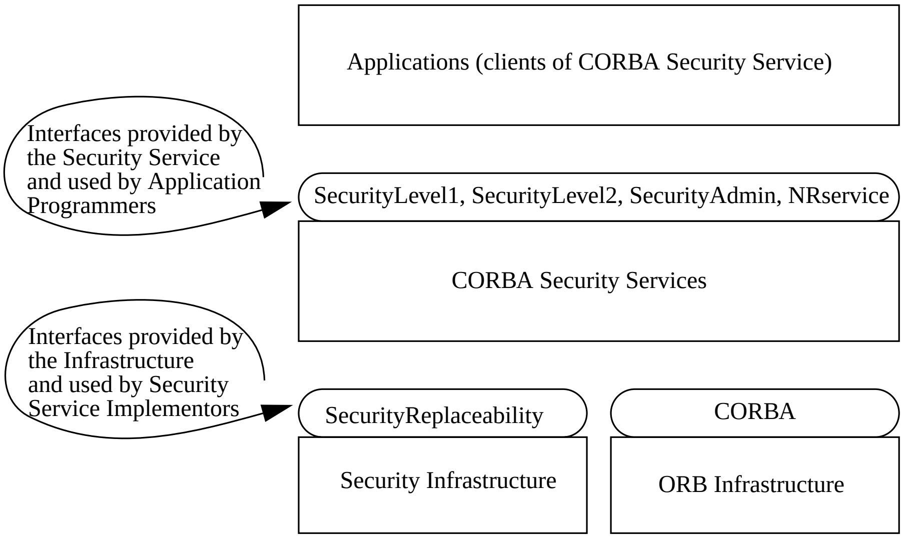

Applications (clients of CORBA Security Service)
Interfaces provided by the Security Service and used by Application Programmers
SecurityLevel1, SecurityLevel2, SecurityAdmin, NRservice
CORBA Security Services
Interfaces provided by the Infrastructure and used by Security Service Implementors
SecurityReplaceability
CORBA
Security Infrastructure
ORB Infrastructure

Figure 1-1 Modules and Their Relation to Layers of the Architecture

The **SecurityReplaceability** module defines the interfaces that must be used, together with certain interfaces defined in the **SecurityLevel2** module, to encapsulate the underlying security infrastructure so as to enable components of the Security Service to use them interchangeably.

Interfaces 2

Contents

This chapter contains the following topics.

2.1 Security Reference Model

This section describes a security reference model that provides the overall framework for CORBA security. The purpose of the reference model is to show the flexibility for defining many different security policies that can be used to achieve the appropriate level of functionality and assurance. As such, the security reference model functions as a guide to the security architecture.

2.1.1 Definition of a Security Reference Model

A reference model describes how and where a secure system enforces security policies. Security policies define:

- Under what conditions active entities (such as clients acting on behalf of users) may access objects.

- What authentication of users and other principals is required to prove who they are, what they can do, and whether they can delegate their rights. (A principal is a human user or system entity that is registered in and is authentic to the system.)
- The security of communications between objects, including the trust required between them and the quality of protection of the data in transit between them.
- What accountability of which security-relevant activities is needed.

Figure 2-1 depicts the model for CORBA secure object systems. All object invocations are mediated by appropriate security functions to enforce policies such as access controls. These functions should be tamper-proof, always be invoked when required by security policy, and function correctly.

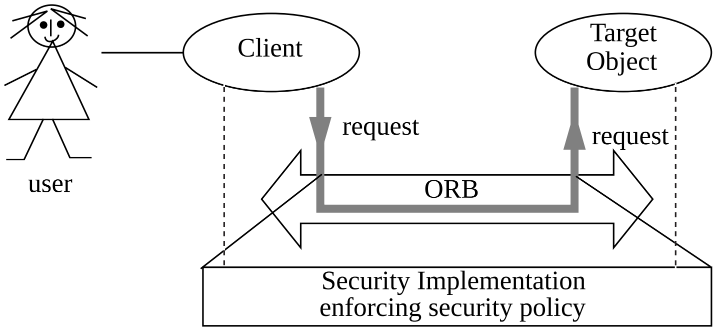


Figure 2-1 A Security Model for Object Systems

Many application objects are unaware of the security policy and how it is enforced. The user can be authenticated prior to calling the application client and then security is subsequently enforced automatically during object invocations. Some applications will need to control or influence what policy is enforced by the system on their behalf, but will not do the enforcement themselves. Some applications will need to enforce their own security, for example, to control access to their own data or audit their own security-relevant activities.

The ORB cannot be completely unaware of security as this would result in insecure systems. The ORB is assumed to at least handle requests correctly without violating security policy, and to call Security Services as required by security policy.

A security model normally defines a *specific* set of security policies. Because the OMG Object Management Architecture (OMA) must support a wide variety of different security policies to meet the needs of many commercial markets, a single instance of a security model is not appropriate for the OMA. Instead, a security reference model is defined that provides a framework for supporting many different kinds of policies. The security reference model is a *meta-policy* because it is intended to encompass all possible security policies supported by the OMA.

The meta-policy defines the abstract interfaces that are provided by the security architecture defined in this document. The model enumerates the security functions that are defined as well as the information available. In this manner, the meta-policy provides guidance on the permitted flexibility of the policy definition. The remaining sections describe the elements of the meta-model. The description is kept deliberately general at this point.

2.1.2 Principals and Their Security Attributes

An active entity must establish its rights to access objects in the system. It must either be a principal, or a client acting on behalf of a principal.

A **principal** is a human user or system entity that is registered in and authentic to the system. **Initiating principals** are the ones that initiate activities. An initiating principal may be **authenticated** in a number of ways, the most common of which for human users is a password. For systems entities, the authentication information such as its long-term key, needs to be associated with the object.

An initiating principal has at least one, and possibly several **identities** (represented in the system by attributes) which may be used as a means of:

- Making the principal accountable for its actions.
- Obtaining access to protected objects (though other privilege attributes of a principal may also be required for access control).
- Identifying the originator of a message.
- Identifying who to charge for use of the system.

There may be several forms of identity used for different purposes. For example, the **audit identity** may need to be anonymous to all but the audit administrator, but the **access identity** may need to be understood so that it can be specified as an entry in an access control list. The same value of the identity can be used for several of the above.

The principal may also have **privilege attributes** which can be used to decide what it can access. A variety of privilege attributes may be available depending on access policies (see Section 2.1.4.3, “Access Policies,” on page 2-9). The privilege attributes, which a principal is permitted to take, are known by the system. At any one time, the principal may be using only a subset of these permitted attributes, either chosen by the principal (or an application running on its behalf), or by using a default set specified for the principal. There may be limits on the duration for which these privilege attributes are valid and may be controls on where and when they can be used.

Security attributes may be acquired in three ways:

1. Some attributes may be available, without authentication, to any principal. This specification defines one such attribute, called *Public*.
2. Some attributes are acquired through authentication; identity attributes and privilege attributes are in this category.
3. Some attributes are acquired through delegation from other principals.

When a user or other principal is authenticated, it normally supplies:

- Its **security name**.

- The **authentication information** needed by the particular authentication method used.
- Requested **privilege attributes** (though the principal may change these later).

A principal's security attributes are maintained in secure CORBA systems in a **credential** as shown in Figure 2-2.

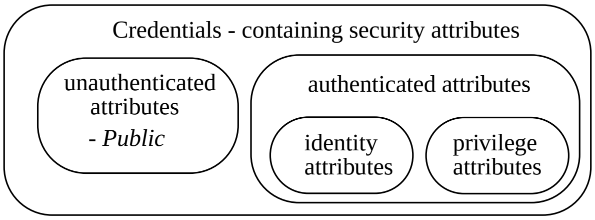


Figure 2-2 Credential Containing Security Attributes

2.1.3 Secure Object Invocations

Most actions in the system are initiated by principals (or system entities acting on their behalf). For example, after the user logs onto the system, the client invokes a target object via an ORB as shown in Figure 2-3.

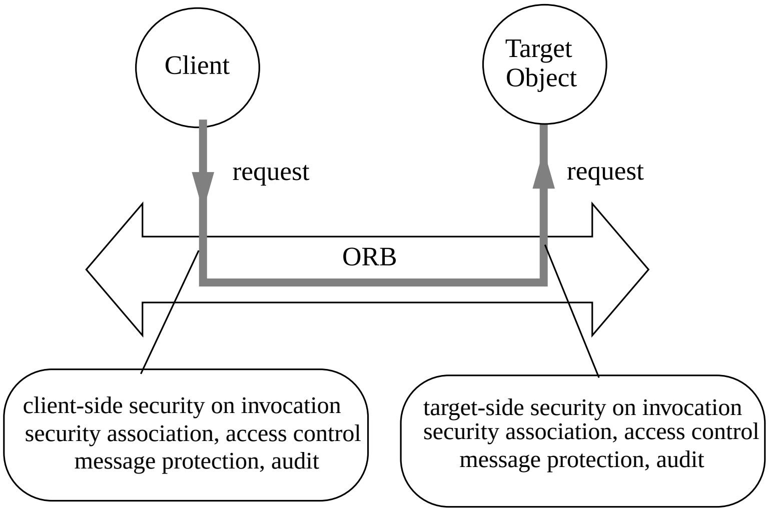

Client
Target Object
request
request
ORB
client-side security on invocation
security association, access control
message protection, audit
target-side security on invocation
security association, access control
message protection, audit

Figure 2-3 Invocation of Target Object via ORB

What security functionality is needed on object invocation depends on security policy. It may include:

- Establishing a **security association** between the client and target object so that each has the required trust that the other is who it claims to be. In many implementations, associations will normally persist for many interactions, not just a single invocation. (Within some environments, the trust may be achieved by local means, without use of authentication and cryptography.)

- Deciding whether this client (acting for this principal) can perform this operation on this object according to the access control policy, as described in Section 2.1.4, “Access Control Model,” on page 2-7.
- Auditing this invocation if required, as described in Section 2.1.5, “Auditing,” on page 2-11.
- Protecting the request and response from modification or eavesdropping in transit, according to the specified quality of protection.

For all these actions, security functions may be needed at the client and target object sides of the invocation. For example, protecting a request may require integrity sealing of the message before sending it, and checking the seal at the target.

The association is asymmetric. If the target object invokes operations on the client, a new association is formed. It is possible for a client to have more than one association with the same target object. The application is unaware of security associations; it sees only requests and responses.

A secure system can also invoke objects in an insecure system. In this case, it will not be possible to establish trust between the systems, and the client system may restrict the requests passed to the target.

2.1.3.1 *Establishing Security Associations*

The client and target object establish a secure association by:

- Establishing trust in one another’s identities, which may involve the target authenticating the client’s security attributes and/or the client’s authenticating the target’s security name.
- Making the client’s credentials (including its security attributes) available to the target object.
- Establishing the security context which will be used when protecting requests and responses in transit between client and target object.

The way of establishing a security association between client and object depends on the security policies governing both the client and target object, whether they are in the same domain, and the underlying security mechanism. For example, the type of authentication and key distribution used.

The security policies define the choice of security association options such as whether one-way or mutual authentication is wanted between client and target, and the quality of protection of data in transit between them.

The security policy is enforced using underlying security mechanisms. This model allows a range of such mechanisms for security associations. For example, the mechanism may use symmetric (secret) key technology, asymmetric (public) key technology, or a combination of these. The Key Distribution services, Certification Authorities and other underlying Security services, which may be used, are not visible in the model.

2.1.3.2 Message Protection

Requests and responses can be protected for:

- Integrity. This prevents undetected, unauthorized modification of messages and may detect whether messages are received in the correct order and if any messages have been added or removed.
- Confidentiality. This ensures that the messages have not been read in transit.

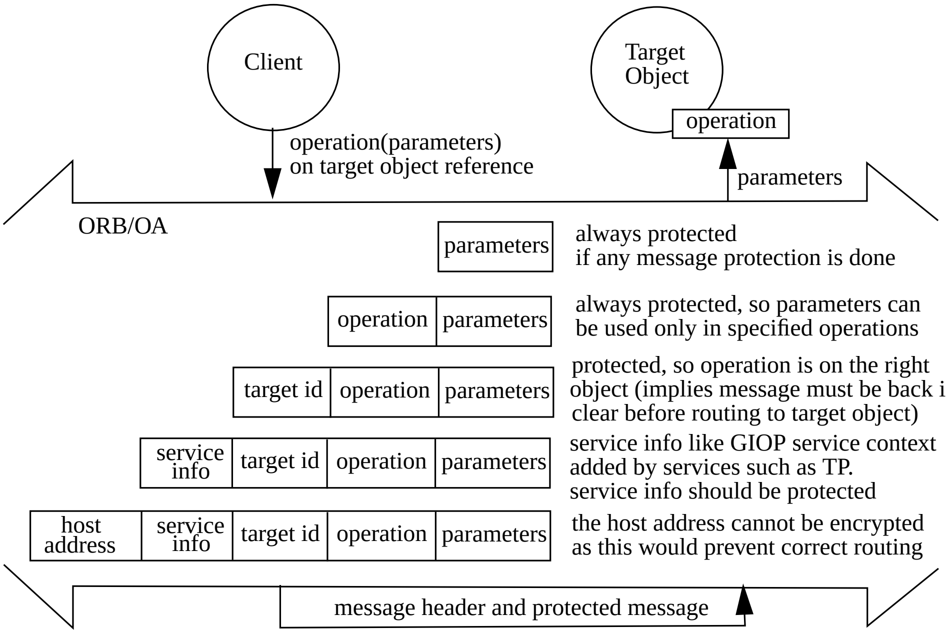
A security association may in some environments be able to provide integrity and confidentiality protection through mechanisms inherent in the environment, and so avoid having to use encryption.

The security policy specifies the strength of integrity and confidentiality protection needed. Achieving this integrity protection may require sealing the message and including sequence numbers. Confidentiality protection may require encrypting it.

This security reference model allows a choice of cryptographic algorithms for providing this protection.

Performing a request on a remote object using an ORB and associated services, such as TP, might cause a message to be constructed to send to the target as shown in Figure 2-4. At the target, this process is reversed, and results in the ORB invoking the operation on the target passing it the parameters sent by the client. The reply returned follows a similar path.

Message protection could be provided at different points in the message handling functionality of an ORB, which would affect how much of the message is protected.



Client
Target Object
operation
operation(parameters) on target object reference
parameters
ORB/OA
parameters
always protected if any message protection is done
operation
parameters
always protected, so parameters can be used only in specified operations
target id
operation
parameters
protected, so operation is on the right object (implies message must be back in clear before routing to target object)
service info
target id
operation
parameters
service info like GIOP service context added by services such as TP. service info should be protected
host address
service info
target id
operation
parameters
the host address cannot be encrypted as this would prevent correct routing
message header and protected message

Figure 2-4 Message Protection

Messages are protected according to the quality of protection required which may be for integrity, but may also be for confidentiality. Both integrity and confidentiality protection are applied to the same part of the message. The request and response may be protected differently.

The CORBA security model can protect messages even when there is no security in the underlying communications software. In this case, the message protected by CORBA security includes the target id, operation and parameters, and any service information included in the message.

In some systems, protection may be provided below the ORB message layer (for example, using the secure sockets layer or even more physical means). In this case, an ORB that knows such security is available will not need to provide its own message protection.

Note that as messages will normally be integrity protected, this will limit the type of interoperability bridge that can be used. Any bridge that changes the protected part of the message after it has been integrity (or confidentiality) protected will cause the security check at the target to fail unless a suitable security gateway is used to re-protect the message.

2.1.4 Access Control Model

The model depicted in Figure 2-5 on page 2-8 provides a simple framework for many different access control security policies. This framework consists of two layers: an object invocation access policy, which is enforced automatically on object invocation, and an application access policy, which the application itself enforces.

The object invocation access policy governs whether this client, acting on behalf of the current principal, can invoke the requested operation on this target object. This policy is enforced by the ORB and the Security services it uses, for all applications, whether they are aware of security or not.

The application object access policy is enforced within the client and/or the object implementation. The policy can be concerned with controlling access to its internal functions and data, or applying further controls on object invocation.

All instantiations of the security reference model place at least some trust in the ORB to enforce the access policy. Even in architectures where the access control mediation occurs solely within the client and target objects, the ORB is still required to validate the request parameters and ensure message delivery as described above.

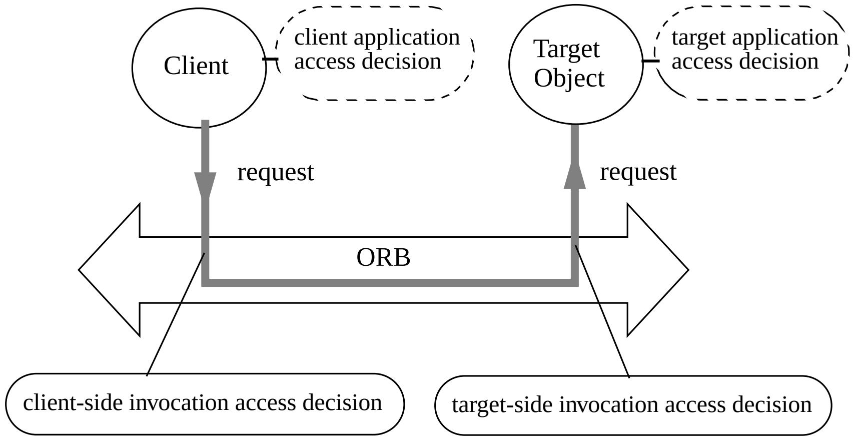

Client
client application access decision
Target Object
target application access decision
request
request
ORB
client-side invocation access decision
target-side invocation access decision

Figure 2-5 Access Control Model

The access control model shows the client invoking an operation as specified in the request, and also shows application access decisions, which can be independent of this.

2.1.4.1 Object Invocation Access Policy

A client may invoke an operation on the target object as specified in the request only if this is allowed by the object invocation access policy. This is enforced by **Access Decision Functions**.

Client side access decision functions define the conditions that allow the client to invoke the specified operation on the target object. Target side access decision functions define the conditions that allow the object to accept the invocation. One or both of these may not exist. Some systems may support target side controls only, and even then, only use them for some of the objects.

The access policy for object invocation is built into these access decision functions, which just provide a yes/no answer when asked to check if access is allowed. A range of access policies can be supported as described in Section 2.3.10, “Access Control,” on page 2-103.

The access decision function used on object invocation to decide whether access is allowed bases its decision on:

- The current privilege attributes of the principal (see Section 2.1.2, “Principals and Their Security Attributes,” on page 2-3). Note that these can include capabilities.
- Any controls on these attributes, for example, the time for which they are valid.
- The operation to be performed.
- The control attributes of the target object (see Section 2.1.4, “Access Control Model,” on page 2-7).

The first three of these functions are available as part of the environment of the object invocation.

The control attributes for the target object are associated with the object when it is created (though may be changed later, if security policy permits).

2.1.4.2 Application Access Policy

Applications may also enforce access policies. An application access policy may control who can invoke the application, extending the object invocation access policy enforced by the ORB, and taking into account other items such as the value of the parameters, or the data being accessed. As for standard object invocation access controls, there may be client and target object access decision functions.

An application object may also control access to finer-grained functions and data encapsulated within it, which are not separate objects.

In either case, the application will need its own access decision function to enforce the required access control rules.

2.1.4.3 Access Policies

The general access control model described here can be used to support a wide range of access policies including **Access Control List** schemes, **label-based** schemes, and **capability** schemes. This section describes the overall authorization model used for all types of access control.

The authorization model is based on the use of access decision functions, which decide whether an operation or function can be performed by applying access control rules using:

- Privilege attributes of the initiator (called initiator Access Control Information or ACI in ISO/IEC 10181-3).
- Control attributes of the target (sometimes known as the target ACI).
- Other relevant information about the action such as the operation and data, and about the context, such as the time.

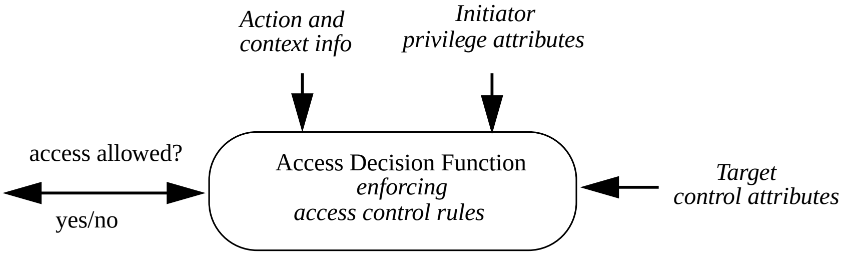


Figure 2-6 Authorization Model

The privilege and control attributes are the main variables used to control access; therefore, the following sections focus on these.

2.1.4.4 Privilege Attributes

A principal can have a variety of privilege attributes used for access control such as:

- The principal’s access identity.
- Roles, which are often related to the user’s job functions.
- Groups, which normally reflect organizational affiliations. A group could reflect the organizational hierarchy, for example, the department to which the user belongs, or a cross-organizational group, which has a common interest.
- Security clearance.
- Capabilities, which identify the target objects (or groups of objects), and their operations on which the principal is allowed.
- Other privileges that an enterprise defines as being useful for controlling access.

In an object system, which may be large, using individual identities for access control may be difficult if many sets of control attributes need to be changed when a user joins or leaves the organization or changes his job. Where possible, controls should be based on some grouping construct (such as a role or organizational group) for scalability.

The security reference model does not dictate the particular privilege attributes, that any compliant secure system must support; however, this specification does define a standard, extensible set of privilege attribute types.

Note – In this specification, *privilege* is often used as shorthand for *privilege attribute.*

2.1.4.5 Control Attributes

Control attributes are associated with the target. Examples are:

- Access control lists, which identify permitted users by name or other privilege attributes, or
- Information used in label-based schemes, such as the classification of an object, which identifies (according to rules) the security clearance of principals allowed to perform particular operations on it.

An object system may have many objects, each of which may have many operations, so it may not be practical to associate control attributes with each operation on each object. This would impose too large an overhead on the administration of the system, and the amount of storage needed to hold the information.

Control attributes are therefore expected to be shared by categories of objects, particularly objects of the same type in the same security policy domain. However, they could be associated with an individual object.

Rights

Control attributes may be associated with a set of operations on an object, rather than each individual operation. Therefore, a user with specified privileges may have **rights** to invoke a specific set of operations.

It is possible to define what rights give access to what operations.

2.1.4.6 Access Policies Supported by This Specification

The model allows a range of access policies using control attributes, which can group subjects (using privileges), objects (using domains), and operations (using rights).

This specification defines a particular access policy type and associated management interface as part of security functionality Level 2. This is defined in DomainAccessPolicy Interface under Section 2.4.4, “Access Policies,” on page 2-119.

Regardless of the access control policy management interface used (i.e., regardless of whether the particular Level 2 access policy interfaces or other interfaces not defined in this specification are used), all access decisions on object invocation are made via a standard access decision interface, so the access control policy can be changed either by administrative action on, or substitution of, the objects that define the policy and implement the access decision. However, different management interfaces will ordinarily be required for management of different types of control attributes.

2.1.5 Auditing

Security auditing assists in the detection of actual or attempted security violations. This is achieved by recording details of **security relevant events** in the system. (Depending on implementation, recording an audit event may involve writing event information to a log, generating an alert or alarm, or some other action.) Audit policies specify which events should be audited under what circumstances.

There are two categories of audit policies: *system audit policies,* which control what events are recorded as the result of relevant system activities, and *application audit policies,* which control which events are audited by applications.

System events, which should be auditable, include events such as authentication of principals, changing privileges, success or failure of object invocation, and the administration of security policies. These system events may occur in the ORB or in security or other services, and these components generate the required audit records.

Application events may be security relevant, and therefore may need auditing depending on the application. For example, an application that handles money transfers might audit who transferred how much money to whom.

Events can be categorized by event family (e.g., system, financial application service), and event type within that family. For example, there are defined event types for system events.

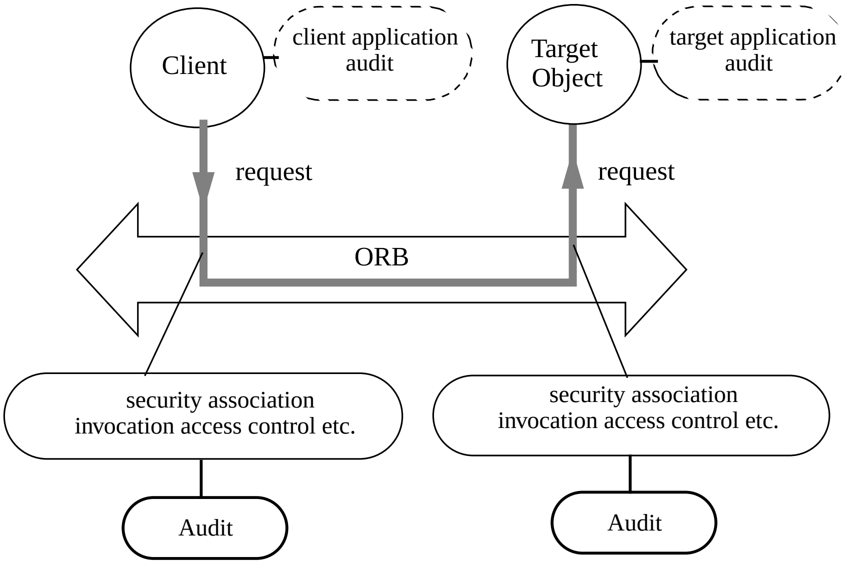

Client
client application audit
Target Object
target application audit
request
request
ORB
security association invocation access control etc.
security association invocation access control etc.
Audit
Audit

Figure 2-7 Auditing Model

Potentially a very large number of events could be recorded; audit policies are used to restrict what types of events to audit under which circumstances. System audit policies are enforced automatically for all applications, even security unaware ones.

The invocation audit policy is enforced at a point in the ORB where the target object and operation for the request are known, and the reply status is known. The model supports audit policies where the decision on whether to audit an event can be based on the event type (such as method invocation complete, access control check done, security association made), the success or failure of this event (only failures may be audited), the object and the operation being invoked, the audit id of principal on whose behalf the invocation is being done, and even the time of day.

This specification defines a particular invocation audit policy type and associated management interfaces as part of security functionality Level 2. This allows decisions on whether to audit an invocation to depend on the object type, operation, event type, and success or failure of this.

The specification also defines a particular audit policy type for application auditing, which allows decisions on whether to audit the event to be based on the event type and its success or failure.

Events can either be recorded on audit trails for later analysis or, if they are deemed to be serious, alarms can be sent to an administrator. Application audit trails may be separate from system ones. This specification includes how audit records are generated and then written to audit channels, but not how these records are filtered later, how audit trails and channels are kept secure, and how the records can be collected and analyzed.

2.1.6 Delegation

In an object system, a client calls on an object to perform an operation, but this object will often not complete the operation itself, so will call on other objects to do so. This will usually result in a chain of calls on other objects as shown in Figure 2-8.

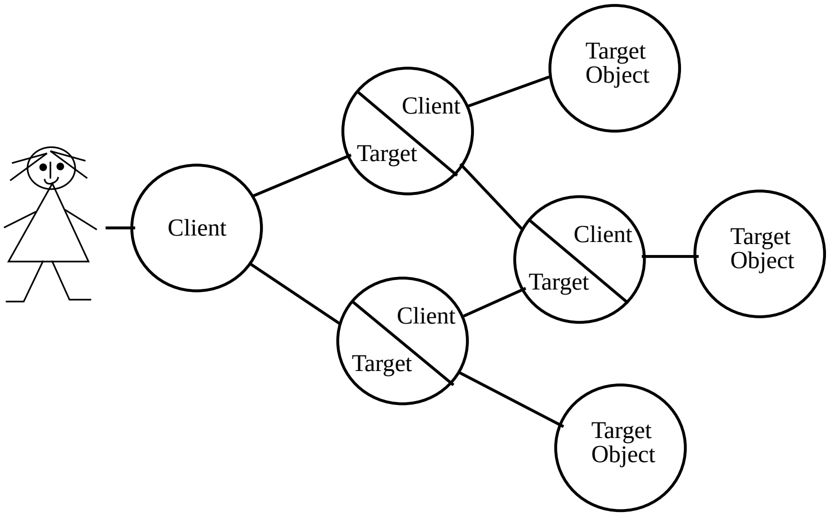


Figure 2-8 Delegation Model

This complicates the access model described in Section 2.1.4, “Access Control Model,” on page 2-7, as access decisions may need to be made at each point in the chain. Different authorization schemes require different access control information to be made available to check which objects in the chain can invoke which further operations on other objects.

In **privilege delegation**, the initiating principal’s access control information (i.e., its security attributes) may be delegated to further objects in the chain to give the recipient the rights to act on its behalf under specified circumstances.

Another authorization scheme is **reference restriction** where the rights to use an object under specified circumstances are passed as part of the object reference to the recipient. Reference restriction is not included in this specification, though described as a potential future security facility in Appendix F, “Facilities Not in This Specification”.

The following terms are used in describing delegation options:

- **Initiator**: the first client in a call chain.
- **Final target**: the final recipient in a call chain.

- **Intermediate**: an object in a call chain that is neither the initiator nor the final target.
- **Immediate invoker**: an object or client from which an object receives a call.

2.1.6.1 Privilege Delegation

In many cases, objects perform operations on behalf of the initiator of a chain of object invocations. In such cases, the initiator needs to delegate some or all of its privilege attributes to the intermediate objects which will act on its behalf.

Some intermediates in a chain may act on their own behalf (even if they have received delegated credentials) and perform operations on other objects using their own privileges. Such intermediates must be (or represent) principals so that they can obtain their own privileges to be transmitted to objects they invoke.

Some intermediates may need to use their own privileges at some times, and delegated privileges at other times.

A target may wish to restrict which of its operations an invoker can perform. This restriction may be based on the identity or other privilege attributes of the initiator. The target may also want to verify that the request comes from an authorized intermediate (or even check the whole chain of intermediates). In these cases, it must be possible to distinguish the privileges of the initiator and those of each intermediate.

Some restrictions may or may not be placed by the initiator about the set of objects which may be involved in a delegation chain.

When no restrictions are placed and only the initiator's privileges are being used, this case is called impersonation.

When restrictions are placed, additional information is used so that objects can verify whether or not their characteristics (e.g., their name or a part of their name) satisfy the restrictions. In order to allow clients or initiating objects to specify this additional information, objects can be (securely) associated with these characteristics (e.g., their name).

2.1.6.2 Overview of Delegation Schemes

There are potentially a large number of delegation models. They can all be captured using the following sentence.

An intermediate invoking a target object may perform:

1. one method on one object
2. several methods on one object
3. any method on: a. one object
   b. some object(s) (target restrictions)
   c. any object (no target restrictions)

	(no privileges	
using	(a subset of the initiator’s privileges	(simple delegation)
	(both the initiator’s and its own privileges	(composite delegation)
	(received privileges and its own privileges	(combined or traced delegation, depending on whether privileges are combined or concatenated)
during some validity period		(part of time constraints)
for a specified number of invocations		(part of time constraints)

When delegating privileges through a chain of objects, the caller does not know which objects will be used in completing the request, and therefore cannot easily restrict privileges to particular methods on objects. It generally relies on the target’s control attributes to do this.

A privilege delegation scheme may provide any of the other controls, though no one scheme is likely to provide all of them.

2.1.6.3 Facilities Potentially Available

Different facilities are available to intermediates (or clients) before initiating object invocations and to intermediate or target objects accepting an invocation.

Controls Used Before Initiating Object Invocations

A client or intermediate can specify restrictions on the use of the access control information provided to another intermediate or to a target object. Interfaces may allow support of the following facilities.

- **Control of privileges delegated**. An initiator (or an intermediate) can restrict which of its own privileges are delegated.
- **Control of target restrictions**. An initiator (or an intermediate) can restrict where individual privileges can be used. This restriction may apply to particular objects, or some grouping of objects. It may restrict the target objects, which may use some privileges for access control, and the intermediates, which can also delegate them.

  Control of privileges used. As previously described, there are several options for deciding which privileges an intermediate object may use when invoking another object. Note that delegated privileges are not actually delegated to a single target object; they are available to any object running under the same identity as the target object in the target object’s address space (since any objects in the target’s address space may retrieve the inbound Credentials and any object sharing the target’s identity may successfully become the caller’s delegate).

  The specified interfaces allow the following.

- **No delegation**

The client permits the intermediate to use its privileges for access control decisions, but does not permit them to be delegated, so the intermediate object cannot use these privileges when invoking the next object in the chain.

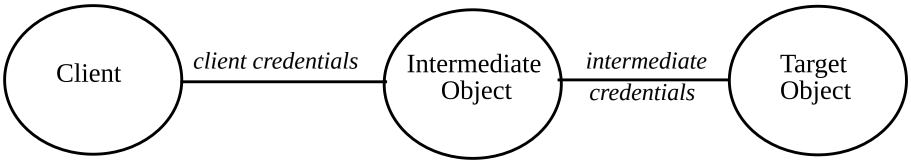


Figure 2-9 No Delegation

- **Simple delegation**

The client permits the intermediate to assume its privileges, both using them for access control decisions and delegating them to others. The target object receives only the client's privileges, and does not know who the intermediate is (when used without target restrictions, this is known as impersonation).

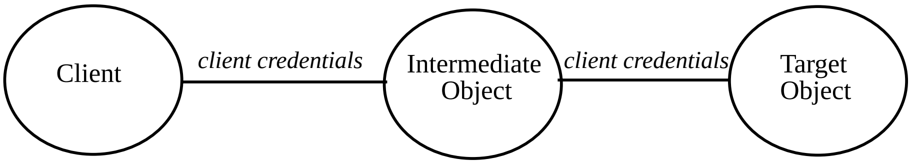


Figure 2-10 Simple Delegation

- **Composite delegation**

The client permits the intermediate object to use its credentials and delegate them. Both the client privileges and the immediate invoker’s privileges are passed to the target, so that both the client privileges and the privileges from the immediate source of the invocation can be individually checked.

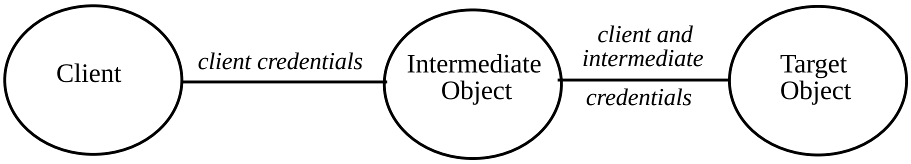


Figure 2-11 Composite Delegation

- **Combined privileges delegation**

The client permits the intermediate object to use its privileges. The intermediate converts these privileges into credentials and combines them with its own credentials. In that case, the target cannot distinguish which privileges come from which principal.

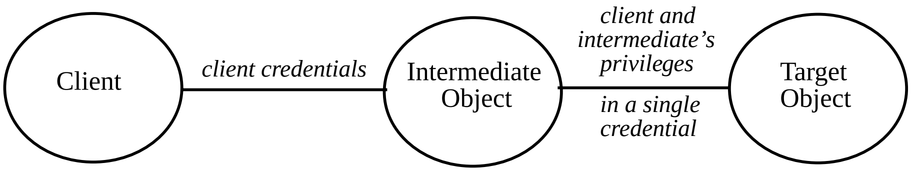


Figure 2-12 Combined Privileges Delegation

- **Traced delegation**

  The client permits the intermediate object to use its privileges and delegate them. However, at each intermediate object in the chain, the intermediate's privileges are added to privileges propagated to provide a trace of the delegates in the chain.

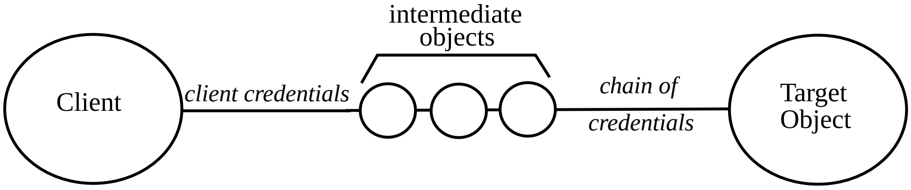


Figure 2-13 Traced Delegation

A client application may not see the difference between the last three options, it may just see them all as some form of "composite" delegation. However, the target object can obtain the credentials of intermediates and the initiator separately if they have been transmitted separately.

- **Control of time restrictions**. Time periods can be applied to restrict the duration of the delegation. In some implementations, the number of invocations may also be controllable.

Facilities Used on Accepting Object Invocations

An intermediate or a target object should be able to:

- Extract received privileges and use them in local access control decisions.
  Often only the privileges of the initiator are relevant. When this is not the case, only the privileges of the immediate invoker may be relevant. In some cases, both are relevant. Finally, the most complex authorization scheme may require the full tracing of the initiator and all the intermediates involved in a call chain.
  In addition, some targets may need to obtain the miscellaneous security attributes (such as audit identity, charging identity) and the associated target restrictions and time constraints.
- Extract credentials (when permitted) for use when making the next call as a delegate.

- Build (when permitted) new credentials from the received access control information with changed (normally reduced) privileges and/or different target restrictions or time constraints.

2.1.6.4 Specifying Delegation Options

The administrator may specify which delegation option should be used by default when an object acts as an intermediate. For example, he may specify whether a particular intermediate object normally delegates the initiating principal's privileges or uses its own, or both if needed. Also, the access policy used at the target could permit or deny access based on more than one of the privileges it received (e.g., the initiator's and the intermediate's). This allows many applications to be unaware of the delegation options in use, as many of the controls for delegation are done automatically by the ORB when the intermediate invokes the next object in the chain.

However, a security-aware intermediate object may itself specify what delegation it wants. For example, it may choose to use the original principal's privileges when invoking some objects and its own when invoking others.

2.1.6.5 Technology Support for Delegation Options

Different security technologies support different delegation models. Currently, no one security technology supports all the options described above.

In Security Functionality Level 1, all delegation is done automatically in the ORB according to delegation policy, so the objects in the chain cannot change the mode of delegation used, or restrict privileges passed and where or when they are used.

Of the options on which credentials are passed, only *no delegation* and *impersonation* (simple delegation without any target restrictions) *need* to be supported.

In Security Functionality Level 2, applications may use any of the interfaces specified, but may get a CORBA::NO_IMPLEMENT exception returned. Note that these interfaces do not allow the application to set controls such as target restrictions. Appendix F, “Facilities Not in This Specification” includes potential future advanced delegation facilities, which include such controls.

2.1.7 Non-repudiation

Non-repudiation services provide facilities to make users and other principals accountable for their actions. Irrefutable evidence about a claimed event or action is generated and can be checked to provide proof of the action. It can also be stored in order to resolve later disputes about the occurrence or the nonoccurrence of the event or action.

The non-repudiation services specified here are under the control of the applications rather than used automatically on object invocation, so are only available to applications aware of this service.

Depending on the non-repudiation policy in effect, one or more pieces of evidence may be required to prove that some kind of event or action has taken place. The number and the characteristics of each depends upon that non-repudiation policy. As an example, evidence containing a timestamp from a trusted authority may be required to validate evidence.

There are many types of non-repudiation evidence, depending on the characteristics of the event or action. In order to distinguish between them, the types are defined and are part of the evidence. Conceptually, evidence may thus be seen as being composed of the following components:

- non-repudiation policy (or policies) applicable to the evidence
- type of action or event
- parameters related to the type of action or event

A date and time are also part of the evidence. This shows when an action or event took place and allows recovery from some situations such as the compromise of a key.

The evidence includes some proof of the origin of data, so a recipient can check where it came from. It also allows the integrity of the data to be verified.

Facilities included here allow an application to deal with evidence of a variety of types of actions or events. Two common types of non-repudiation evidence are the evidence of proof of creation of a message and proof of receipt of a message.

Non-repudiation of Creation protects against an originator's false denial of having created a message. It is achieved at the originator by constructing and generating evidence of Proof of Creation using non-repudiation services. This evidence may be sent to a recipient to verify who created the message, and can be stored and then made available for subsequent evidence retrieval.

Non-repudiation of Receipt protects against a recipient's false denial of having received a message (without necessarily seeing its content). It is achieved at the recipient by constructing and generating evidence of Proof of Receipt using the non-repudiation services. This is shown in Figure 2-14.

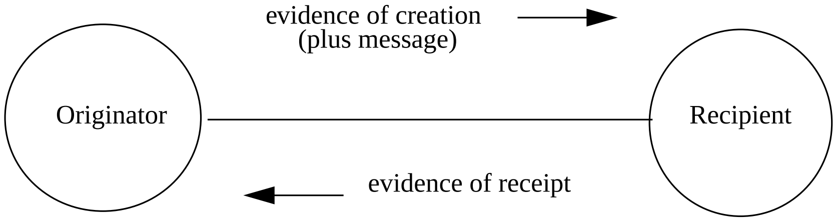


Figure 2-14 Proof of Receipt

One or more Trusted Third Parties need to be involved, depending on the choice of mechanism or policy.

Non-repudiation services may include:

- Facilities to generate evidence of an action and verify that evidence later.

- A delivery authority which delivers the evidence (often with the message) from the originator to the recipient. Such a delivery authority may generate *proof of origin* (to protect against a sender's false denial of sending a message or its content) and *proof of delivery* (to protect against a recipient's false denial of having received a message or its content). Non-repudiation of Origin and Delivery are defined in ISO 7498-2.
- An evidence storage and retrieval facility used when a dispute arises. An adjudicator service may be required to settle the dispute, using the stored evidence.

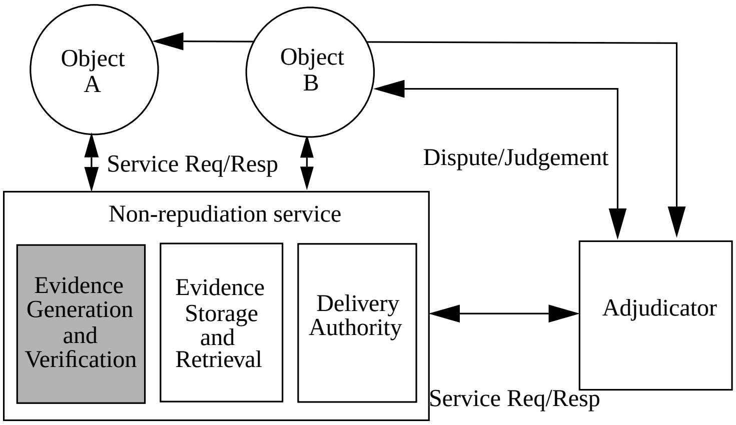


Figure 2-15 Non-repudiation Services

The non-repudiation services illustrated in Figure 2-15 are based on the ISO non-repudiation model; as the shaded box in the diagram indicates, this specification supports only Evidence Generation and Verification, which provides:

- Generation of evidence of an action.
- Verification of evidence of an action.
- Generation of a request for evidence related to a message sent to a recipient.
- Receipt of a request for evidence related to a message received.
- Analysis of details of evidence of an action.
- Collection of the evidence required for long term storage. In this case, more complete evidence may be needed.

The Non-repudiation Service allows an application to deal with a variety of types of evidence, not just the non-repudiation of creation and receipt previously described.

No Non-repudiation Evidence Delivery Authority is defined by this specification; it is anticipated that vendors will want to customize these authorities (which are responsible for delivering messages and related non-repudiation evidence securely in accordance with specific non-repudiation policies) to meet specialized market requirements. Also, no evidence storage and retrieval services are specified, as other object services can be used for this.

Note that this specification does not provide evidence that a request on an object was successfully carried out; it does not require use of non-repudiation within the ORB.

2.1.8 Domains

A domain (as specified in the ORB Interoperability Architecture) is a distinct scope, within which certain common characteristics are exhibited and common rules observed. There are several types of domain relevant to security:

- Security policy domain. The scope over which a security policy is enforced. There may be subdomains for different aspects of this policy.
- Security environment domain. The scope over which the enforcement of a policy may be achieved by some means local to that environment, so does not need to be enforced within the object system. For example, messages will often not need cryptographic protection to achieve the required integrity when being transferred between objects in the same machine.
- Security technology domain. Where common security mechanisms are used to enforce the policies.

These can be independent of the ORB technology domains.

2.1.8.1 Security Policy Domains

A **security policy domain** is a set of objects to which a security policy applies for a set of security related activities and is administered by a **security authority**. (Note that this is often just called a security domain.) The objects are the domain members. The policy represents the rules and criteria that constrain activities of the objects to make the domain secure. Security policies concern access control, authentication, secure object invocation, delegation and accountability. An access control policy applies to the security policies themselves, controlling who may administer security-relevant policy information.

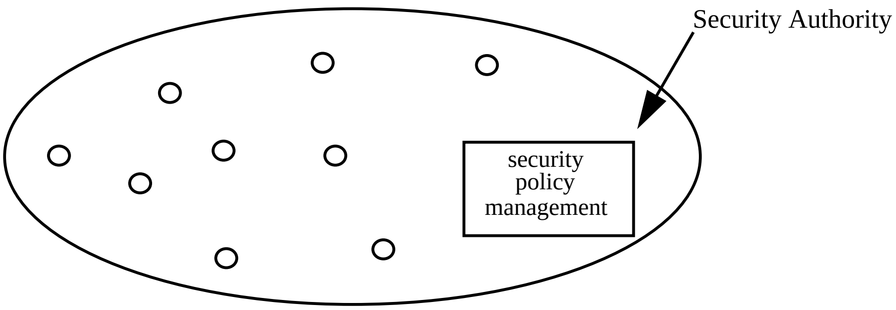


Figure 2-16 Security Policy Domains

Security policy domains provide leverage for dealing with the problem of scale in security policy management (by allowing application of policy at a domain granularity rather than at an individual object instance granularity).

Security policy domains permit application of security policy information to security-unaware objects without requiring changes to their interfaces (by associating the security policy management interfaces with the domain rather than with the objects to which policy is applied).

Domains provide a mechanism for delimiting the scope of administrators' authorities.

Policy Domain Hierarchies

A security authority must be identifiable and responsible for defining the policies to be applied to the domain, but may delegate that responsibility to a number of subauthorities, forming subdomains where the subordinate authorities' policies are applied.

Subdomains may reflect organizational subdivisions or the division of responsibility for different aspects of security. Typically, organization-related domains will form the higher-level superstructure, with the separation of different aspects of security forming a lower-level structure.

For example, there could be:

- An enterprise domain, which sets the security policy across the enterprise.
- Subdomains for different departments, each consistent with the enterprise policy but each specifying more specific security policies appropriate to that department.

With each department, authority may be further devolved:

- Authority for auditing could be the preserve of an audit administrator.
- Control of access to a set of objects could be the responsibility of a specific administrator for those objects.

This supports what is recognized as good security practice (it separates administrators' duties) while reflecting established organizational structures.

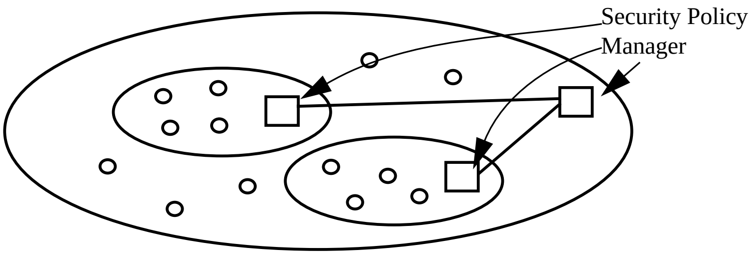


Figure 2-17 Policy Domain Hierarchies

Federated Policy Domains

As well as being structured into superior/subordinate relationships, security policy domains may also be federated. In a federation, each domain retains most of its authority while agreeing to afford the other limited rights. The federation agreement records:

- The rights given to both sides, such as the kind of access allowed.
- The trust each has in the other.

It includes an agreement as to how policy differences are handled, for example, the mapping of roles in one domain to roles in the other.

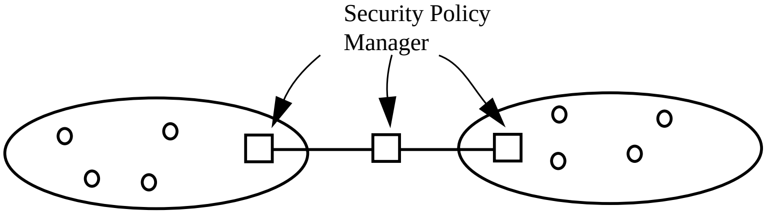


Figure 2-18 Federated Policy Domains

System- and Application-Enforced Policies

In a CORBA system, the “system” security policy is enforced by the distributed ORB and the Security services it uses and the underlying operating systems that support it. This is the only policy that applies to objects unaware of security.

The application security policy is enforced by application objects, which have their own security requirements. For example, they may want to control access to their own functions and data at a finer granularity than the system security policy provides.

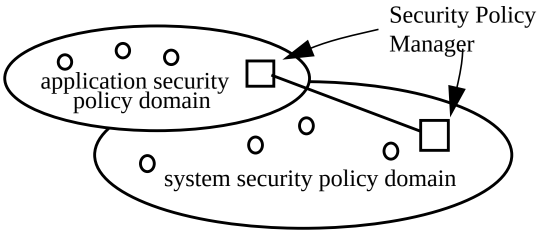

Security Policy Manager
application security policy domain
system security policy domain

Figure 2-19 System- and Application-enforced Policies

Overlapping Policy Domains

Not all policies have the same scope. For example, an object may belong to one domain for access control and a different domain for auditing.

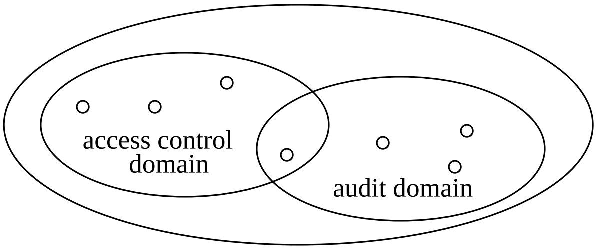


Figure 2-20 Overlapping Policy Domains

In some cases, there may even be overlapping policies of the same type (however, this specification does not require implementations to support overlapping policy domains of the same type).

2.1.8.2 Security Environment Domains

Security policy domains specify the scope over which a policy applies. Security environment domains are the scope over which the enforcement of the policies may be achieved by means local to the environment. The environment supporting the object system may provide the required security, and the objects within a specific environment domain may trust each other in certain ways. Environment domains are by definition implementation-specific, as different implementations run in different types of environments, which may have different security characteristics.

Environment domains are not visible to applications or Security services.

In an object system, the cost of using the security mechanisms to enforce security at the individual object level in all environments would often be prohibitive and unnecessary. For example:

- Preventing objects from interfering with each other might require them to execute in separate system processes or virtual machines (assuming the generation procedure could not ensure this protection) but, in most object systems, this would be considered an unacceptable overhead, if applied to each object.
- Authenticating every object individually could also impose too large an overhead, particularly where:
  - There is a large object population.
  - There is high connectivity, and therefore a large number of secure associations.
  - The object population is volatile, requiring objects to be frequently introduced to the Security services.

This cost can be reduced by identifying security environment domains where enforcement of one or more policies is not needed, as the environment provides adequate protection. Two types of environment domains are considered:

1. **Message protection domains**. These are domains where integrity and/or confidentiality is available by some specific means, for example, an underlying secure transport service is used. An ORB, which knows such protection exists, can exploit it, rather than provide its own message protection.

2. **Identity domains**. Objects in an identity domain can share the same identity. Objects in the same identity domain:
   - when invoking each other, do not need authentication to establish who they are communicating with.
   - are equally trusted by others to handle credentials received from a client. For example, if a client is prepared to delegate its rights to one object in the domain, it is prepared to delegate the same rights to all of them. If any object in the identity domain invokes a further object, that target object is prepared to trust the calling object based on the identity of its identity domain.

Note that neither of these affect what access controls apply to the object (except in that if trust is required and is not established with this domain, then access will be denied).

2.1.8.3 Security Technology Domains

These are domains that use the same security technology for enforcing the security policy. For example:

- The same methods are available for principal authentication and the same Authentication services are used.
- Data in transit is protected in the same way, using common key distribution technology with identical algorithms.
- The same types of access control are used. For example, a particular domain may provide discretionary access control using ACLs using the same type of identity and privilege attributes.
- The same audit services are used to collect audit records in a consistent way.

A particular security technology is normally used to authenticate principals and to form security associations between client and object and handle message protection. (Different technologies may be able to use the same privilege attributes, for example, the same access id and also the same audit id.) An important part of this is the security technology used for key distribution. There are two main types of security technology used for key distribution, both of which are available in commercial products:

- Symmetric key technology where a shared key is established using a trusted Key Distribution Service.
- Asymmetric (or "public") key technology where the client uses the public key of the target (certified by a Certification Authority), while the target uses a related private key.

Public key technology is also the most convenient technology upon which to implement non-repudiation, which has led to its use in several electronic mail products.

The CORBA security interfaces specified here are security mechanism neutral, so they can be implemented using a wide variety of security mechanisms and protocols.

2.1.8.4 Domains and Interoperability

Interoperability between objects depends on whether they are in the same:

- Security technology domain
- ORB technology domain
- Security policy domains
- Naming and other domains

The level of security interoperability fully defined in this CORBA security specification is limited, though it includes an architecture that allows further interoperability to be added.

The following diagram shows a framework of domains and is used to discuss the interoperability goals of this specification.

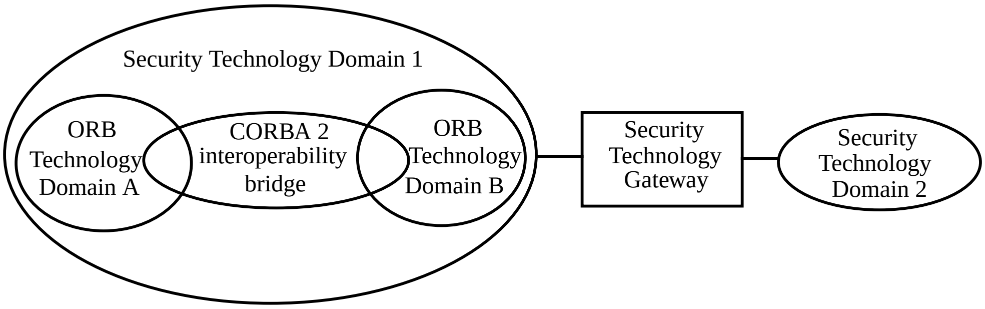

Security Technology Domain 1
ORB Technology Domain A
CORBA 2 interoperability bridge
ORB Technology Domain B
Security Technology Gateway
Security Technology Domain 2

Figure 2-21 Framework of Domains

Interoperating between Security Technology Domains

Sending a message across the boundary between two different security technology domains is only possible if:

- The communication between the objects does not need to be protected, so security is not used between them, or
- A security technology gateway has been provided, which allows messages to pass between the two security technology domains. A gateway could be as simple as a physically secure link between the domains and an agreement between the administrators of the two domains to turn off security on messages sent over the link. On the other hand, it could be a very complicated affair including a protocol translation service with complicated key management logic, for example.

It is not a goal of this specification to define interoperability across Security Technology Domains, and hence to specify explicit support for security technology gateways. This is mainly because the technology is immature and appropriate common technology cannot yet be identified. However, where the security technology in the domains can support more than one security mechanism, this specification allows an appropriate matching mechanism to be identified and used.

Interoperating between ORB Technology Domains

If different ORB implementations are in the same security technology domain, they should be able to interoperate via a CORBA 2 interoperability bridge. However, there may still be restrictions on interoperability when:

- The objects are in different security policy domains, and the security attributes controlling policy in one domain are not understood or trusted in the other domain. As previously described, crossing a security policy boundary can be handled by a security policy federation agreement. This can be enforced in either domain or by a gateway.
- The ORBs are in different naming or other domains, and messages would normally be modified by bridges outside the trusted code of either ORB environment. Security protection prevents tampering with the messages (and therefore any

changes to object references in them). In general, crossing of such domains without using a Security Technology gateway is not possible if policy requires even integrity protection of messages.

2.1.9 Security Management and Administration

Security administration is concerned with managing the various types of domains and the objects within them.

2.1.9.1 Managing Security Policy Domains

For security policy domains, the following is required:

- Managing the domains themselves - creating, deleting them including controlling where they fit in the domain structure.
- Managing the members of the domain, including moving objects between domains.
- Managing the policies associated with the domains - setting details of the security policies as well as specifying which policies apply to which domains.

This specification focuses on management of the security policies. However, managing policy domains and their members in general are expected to be part of the Management Common Facilities, so only an outline specification is given here.

This specification includes a framework for administering of security policies, and details of how to administer particular types of policy. For example, it includes operations to specify the default quality of protection for messages in this domain, the policy for delegating credentials, and the events to be audited.

General administration of all access control policies is not detailed, as the way of administering access control policies is dependent on the type of policy. For example, different administration is needed for ACL-based policies and label-based policies. However, the administration of the standard **DomainAccessPolicy** is defined.

Access policies may use *rights* to group operations for access control. Administration of the mapping of rights to operations is included in this specification. Such mapping of rights to operations is used by the standard **DomainAccessPolicy**, and can also be used by other access policies.

Interfaces for federation agreements allowing interaction with peer domains is left to a later security specification.

2.1.9.2 Managing Security Environment Domains

For environment domains, an administrator may have to specify the characteristics of the environment and which objects are members of the domain. This will often be done in an environment-specific way; therefore, no management interfaces for it are specified here.

2.1.9.3 Managing Security Technology Domains

For security technology domains, administration may include:

- Setting up and maintaining the underlying Security services required in the domain.
- Setting up and maintaining trust between domains in line with the agreements between their management.
- Administering entities in the way required by this security technology. Entities to be administered include principals, which have identities, long-term keys, and optionally privileged attributes.

Such administration is often security technology specific. Also, it may be done outside the object system, as it is a goal of this specification to allow common security technology to be used, and even allow a single user logon to object, as well as other applications. This specification does not include such security technology specific administration.

2.1.10 Implementing the Model

This reference model is sufficiently general to cover a very wide variety of security policies and application domains to allow conformant implementations to be provided to meet a wide variety of commercial and government secure systems in terms of both security functionality and assurance. (Any implementation of this model will need to identify the particular security policies it supports.)

The model also allows different ways of putting together the trusted core of a secure object system to address different requirements. There are a number of implementation choices on how to ensure that the security enforcement cannot be bypassed. This enforcement could be performed by hardware, the underlying operating system, the ORB core, or ORB services. Appendix E, “Guidelines for a Trustworthy System” describes some of these options. (It is important when instantiating this architecture for a particular ORB product, or set of Security services supporting one or more ORBs, to identify what portions of the model must be trusted for what. This should be included in a conformance statement as described in Appendix D).

2.2 Security Architecture

This section explains how the security model is implemented. It describes the complete architecture as needed to support all feature packages described in Section 1.2.2, “CORBA Security and Secure Interoperability Feature Packages,” on page 1-10. Not all of these packages are mandatory for all implementors to support. See Appendix D, “Conformance Details” for a definitive statement of conformance requirements.

This section starts by reviewing the different views that different users have of security in CORBA-compliant systems, as the security architecture must cater to these.

The structural model for security in CORBA-compliant systems is described. This includes some expansion of the ORB service concept introduced into CORBA 2 to support interoperability between ORBS.

The security object models for the three major views (application development, administration, and object system implementors) are then described.

2.2.1 Different Users' View of the Security Model

The security model can be viewed from the following users' perspectives:

- Enterprise management
- The end user
- The application developer
- Administration of an operational system
- The object system implementors

2.2.1.1 Enterprise Management View

Enterprise management are responsible for business assets including IT systems; therefore they have ultimate responsibility for protecting the information in the system. The enterprise view of security is therefore mainly about protecting its assets against perceived threats at an affordable cost. This requires assessing the risks to the assets and the cost of countermeasures against them as described in Appendix E, "Guidelines for a Trustworthy System". It will require setting a security policy for protecting the system, which the security administrators can implement and maintain.

Not all parts of an enterprise require the same type of protection of their assets. Enterprise management may identify different domains where different security policies should apply. Managers will need to agree how much they trust each other and what access they will provide to their assets. For example, when a user in domain A accesses objects in domain B, what rights should he have? One enterprise may also interwork with domains in other enterprises.

Enterprise management therefore knows about the structure of the organization and the security policies needed in different parts of it. Security policy options supported by the model include:

- A choice of access control policies. For example, controls can be based on job roles (or other attributes) and use ACL, capabilities, or label-based access controls.
- Different levels of auditing so choosing which events to be logged can be flexibly chosen to meet the enterprise needs.
- Different levels of protection of information communicated between objects in a distributed system. For example, integrity only or integrity plus confidentiality.

The enterprise manager is not a direct user of the CORBA security system.

2.2.1.2 End User View

The human user is an individual who is normally authenticated to the system to prove who he or she is.

The user may take on different job roles which allow use of different functions and data, thereby allowing access to different objects in the system. A user may also belong to one or more groups (within and across organizations) which again imply rights to access objects. A user may also have other privileges such as a security clearance that permits access to secret documents, or an authorization level that allows the user to authorize purchases of a given amount.

The user is modeled in the system as an initiating principal who can have privilege attributes such as roles and groups and others privileges valid to this organization.

The user invokes objects to perform business functions on his behalf, and his privilege attributes are used to decide what he can access. His audit identity is used to make him individually accountable throughout the system. He has no idea of what further objects are required to perform the business function.

The user view is described further in the security model in Section 2.1, "Security Reference Model," on page 2-1.

2.2.1.3 Application Developer View

The application developer is responsible for the business objects in the system: the applications. His main concern is the business functions to be performed.

Many application developers can be unaware of the security in the system, though their applications are protected by it. So much of the security in the system is hidden from the applications. ORB security services are called automatically on object invocation, and both protect the conversation between objects and control who can access them.

Some application objects need to enforce some security themselves. For example, an application might want to control access based on the value of the data and the time as well as the principal who initiated the operation. Also, an application may want to audit particular security relevant activities.

The model includes a range of security facilities available for those applications that want to use them. For example:

- The quality of protection for object invocations can be specified and used to protect all communication with a particular target or just selected invocations.
- Audit can also be used independently of other security facilities and does not require the application to understand other security issues.
- Other functions, such as user authentication or handling privilege attributes for access control generally require more security understanding and operations on the objects, which represent the user in the system. However, this is still done via generic security interfaces, which hide the particular security technology used.

One special type of application developer is also catered for. The "application" that provides the user interface (user sponsor or logon client) needs an authentication interface capable of fitting with a range of authentication devices. However, the model also allows authentication to be done before calling the object system.

The application view is described in Section 2.3, “Application Developer’s Interfaces,” on page 2-71.

2.2.1.4 Administrator’s View

Administrators, like any other users, know about their job roles and other privileges, and expect these to control what they can do. In many systems, there will be a number of different administrators, each responsible for administering only part of the system. This may be partly to reduce the load on individual administrators, but partly for security reasons, for example to reduce the damage any one person can do.

Administrators and administrative applications see more of the system than other users or normal application developers. For example, the application developers see individual objects whereas the administrator knows how these are grouped, for example, in policy domains.

In an operational system, administrators will be responsible for creating and maintaining the domains, specifying who should be members of the domain, its location, etc. They will also be responsible for administering the security policies that apply to objects in these domains.

An administrator may also be responsible for security attributes associated with initiating principals such as human users, though this may be done outside the object system. This would include administration of privilege attributes about users, but might also include other controls. For example, they might constrain the extent to which the user’s rights can be delegated.

The model does not include explicit management interfaces for managing domains or security attributes of initiating principals, though it does describe the resultant information. Note that the security facilities described here are also applicable to management. For example, management information needs to be protected from unauthorized access and protected for integrity in transit, and significant management actions, particularly those changing security information, need to be audited.

The administrator’s view is further described in Section 2.4, “Administrator’s Interfaces,” on page 2-116.

2.2.1.5 Object System Implementor’s View

Secure object system developers must put together:

- An ORB.
- Other Object Services and/or Common Facilities.
- The security services these require to provide the security features.

The system must be constructed in such a way as to make it secure.

The ORB implementor in a secure object system may use ORB Security services during object invocation, as defined in Section 2.2.2, “Structural Model,” on page 2-32. In addition, protection boundaries are required to prevent interference between objects and will need controlling by the ORB and associated Object Adapter and ORB services.

Certain interfaces are identified as **Locality Constrained**. These interfaces are intended to be accessible only from within the context (e.g., process or RM-ODP capsule) in which they are instantiated (i.e., from within the protection boundary around that context). No object reference to these interfaces can therefore be passed meaningfully outside of that context. The exact details of how this protection boundary is implemented is an implementation detail that the implementor of the service will need to provide in order to establish that the implementation is secure. Locality constrained objects may not be accessible through the DII/DSI facilities, and they may not appear in the Interface Repository. Any attempt to pass a reference to a locality constrained object outside its locality, or any attempt to externalize it using **ORB::object_to_string** will result in the raising of the CORBA::NO_MARSHAL exception.

Object Service and Common Facilities developers may need to be security aware if they have particular security requirements (for example, functions whose use should be limited or audited). However, like any application objects, most should depend on the ORB and associated services to provide security of object invocations.

The Security services implementor has to provide ORB Security services (for security of object invocations) and other security services to support applications’ view of security as previously defined. The ORB Security services implementor shares some application visible security objects such as a principal’s credentials, and also sees the security objects used in making security associations. The Security services should use the Security Policy and other security objects defined in this model to decide what security to provide.

While these security objects may provide all the security required themselves, they will often call on external security services, so that consistent security can be provided for both object and other systems. The Security services defined in this specification are designed to allow for convenient implementation using generic APIs for accessing external security services so it is easier to link with a range of such services. Use of such external security services may imply use of existing, nonobject databases for users, certificates, etc. Such databases may be managed outside the object system.

The Implementor’s view is specified in Section 2.5, “Implementor’s Security Interfaces,” on page 2-143. The implications of constructing the system securely to meet threats are described in Appendix E, “Guidelines for a Trustworthy System”.

2.2.2 Structural Model

The architecture described in this section sets the major concepts on which the subsequent specifications are based.

The structural model has four major levels used during object invocation:

1. Application-level components, which may or may not be aware of security;
2. Components implementing the Security services, independently of any specific underlying security technology. (This specification allows the use of an isolating interface between this level and the security technology, allowing different security technologies to be accommodated within the architecture.) These components are:
    - The ORB core and the ORB services it uses.
    - Security services.
    - Policy objects used by these to enforce the Security Policy.
3. Components implementing specific security technology.
4. Basic protection and communication, generally provided by a combination of hardware and operating system mechanisms.

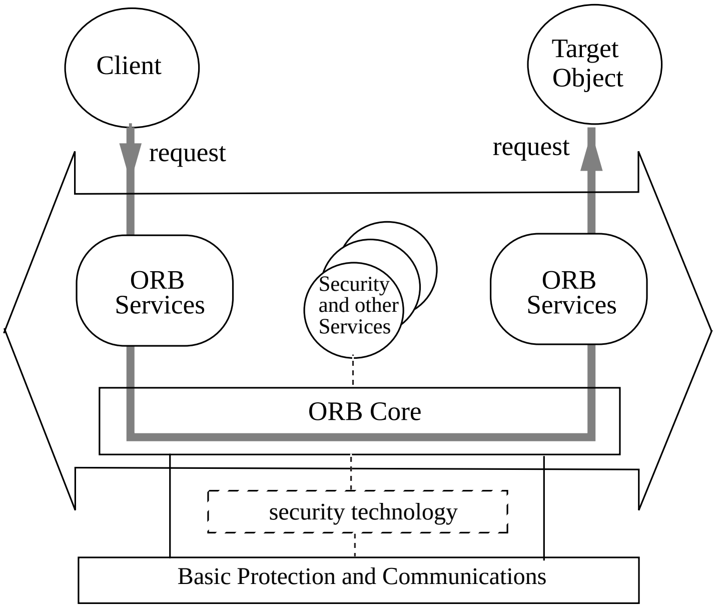


Figure 2-22 Structural Model

Figure 2-22 illustrates the major levels and components of the structural model, indicating the relationships between them. The basic path of a client invocation of an operation on a target object is shown.

2.2.2.1 *Application Components*

Many application components are unaware of security and rely on the ORB to call the required security services during object invocation. However, some applications enforce their own security and therefore call on security services directly (see The Model as Seen by Applications, under Section 2.2.5, “Security Object Models,” on page 2-41). As in the OMA, the client may, or may not, be an object.

2.2.2.2 ORB Services

The ORB Core is defined in the CORBA architecture as “that part of the ORB that provides the basic representation of objects and the communication of requests.” The ORB Core therefore supports the minimum functionality necessary to enable a client to invoke an operation on a target object, with the distribution transparencies required by the CORBA architecture.

An object request may be generated within an implicit context, which affects the way in which it is handled by the ORB, though *not* the way in which a client makes the request. The implicit context may include elements such as transaction identifiers, recovery data and, in particular, security context. All of these are associated with elements of functionality, termed ORB Services, additional to that of the ORB Core but, from the application view, logically present in the ORB.

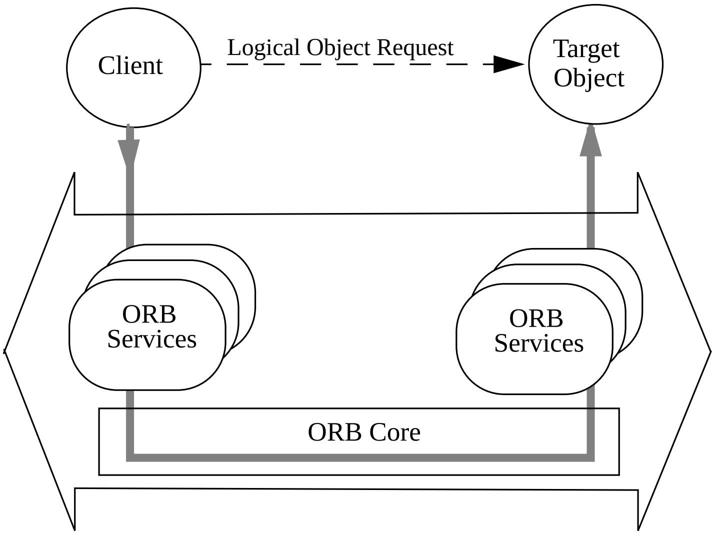


Figure 2-23 ORB Services

Selection of ORB Services

The ORB Services used to handle an object request are determined by:

- The security policies that apply to the client and target object because of the domains to which they belong, for example the access policies, default quality of protection.
- Other static properties of the client and target object such as the security mechanisms and protocols supported.
- Dynamic attributes, associated with a particular thread of activity or invocation; for example, whether a request has integrity or confidentiality requirements, or is transactional.

A client's ORB determines which ORB Services to use at the client when invoking operations on a target object. The target's ORB determines which ORB Services to use at the target. If one ORB does not support the full set of services required, then either the interaction cannot proceed or it can only do so with reduced facilities, which may be agreed to by a process of negotiation between ORBs.

Bindings and Object References at the Client

Before a client can use an object reference to invoke an operation of the target object in a secure way, a security association needs to be established associating the client to the target object, through the particular object reference. This security association is sometimes referred to as the **binding**. The creation and life-style of bindings are implicitly managed by the ORBs and hence the only invariant that one can depend on is that a binding is established before an invocation takes place.

The ORB determines how to establish the binding using the policies, static properties, and dynamic properties associated with the client and target. At the client, the client environment together with an object reference of the target object has associated with it, those policies and static properties of the target object (e.g., the quality of protection needed) that affect how the client's ORB establishes a binding to the object.

Associated with each binding is information specific to the particular usage by the client of the object reference. A binding is uniquely associated with:

- Each object reference of the target object that is held by the client.
- State information that is unique to the association between the target object and the client through the specific object reference (e.g., access policy domain, security context).
- An ORB instance in a process or capsule (c.f. RM-ODP[15]) in which the client is located.

A binding is distinct from the target object, though uniquely associated with it through the object reference. The lifetime of a binding is limited to that of the process or capsule that it is associated with, though it may be shorter (e.g., when the object reference to the target object is destroyed, the binding associated with the object reference is also destroyed).

There is state information associated with the binding at both the client and the server ends. This state information is local to the process or capsule in which the client and the server reside, and its lifetime is the same as that of the binding. The state associated with a binding is not accessible on the client side, since the implicitness of the binding and the uncertainty about its life-style makes such information of questionable value anyway. On the server side, some of this information is accessible through operations of the **Current** object.

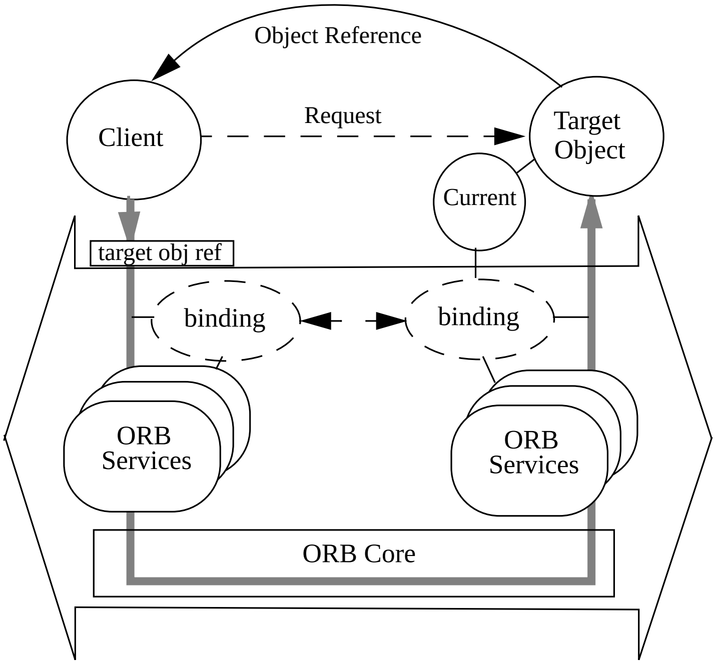


Figure 2-24 Object Reference

If a client requires to invoke operations of the same target object with different invocation policies, it can do so by using the **Object::set_policy_overrides** operation to create new object references with the desired policies (that differ from those associated with the client's environment through the **Current** object) installed as overrides, and then use those new object references to carry out the invocations,

2.2.2.3 *Security Services*

In a secure object system, the ORB Services called will include ORB Security Services for secure invocation and access control.

ORB Security Services and applications may call on underlying security mechanisms for authentication, access control, audit, non-repudiation, and secure invocations. These security services form the Security Replaceability packages.

2.2.2.4 *Security Policies and Domain Objects*

A security policy domain is the set of objects to which common security policies apply as described in Security Policy Domains, under Section 2.1.8, "Domains," on page 2-21. The domain itself is not an object. However, there is a policy domain manager for each security policy domain. This domain manager is used when finding and managing the policies that apply to the domain. The ORB and security services use these to enforce the security policies relevant to object invocation.

When an object reference is created by the ORB, it implicitly associates the object reference with one or more Security Policy domains as described in Administrative Model, under Section 2.2.5, “Security Object Models,” on page 2-41. An implementation may allow object references to be moved between domains later. Since the only way to access objects is through object references, associating object references with policy domains and associated policies, implicitly associates the said policies with the object associated with the object reference. Care should be taken by the applications that is creating object references using **POA** operations (See the Portable Object Adaptor chapter of the *Common Object Request Broker: Architecture and Specification*) to ensure that object references to the same object are not created by the server of that object with different domain associations.

There may be several security policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with each policy domain. (See Section 2.2.5.2, “Administrative Model,” on page 2-58, for a list of policy types.) These policy objects are shared between objects in the domain, rather than being associated with individual objects. (If an object needs to have an individual policy, then there must be a domain manager for it.)

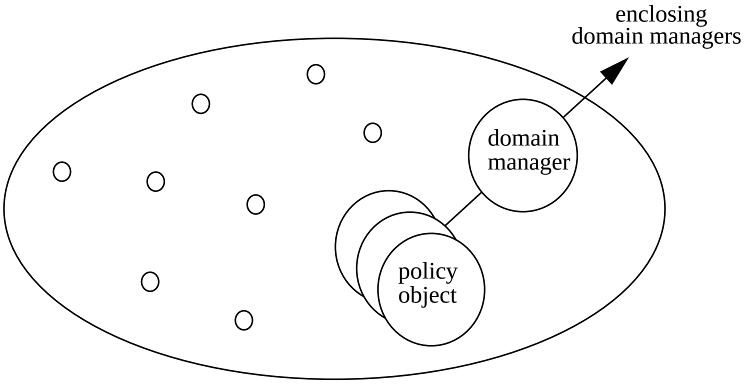


Figure 2-25 Domain Objects

Where an object reference is a member of more than one domain, for example, there is a hierarchy of domains, the object reference is governed by all policies of its enclosing domains. The domain manager can find the enclosing domain’s manager to see what policies it enforces.

The reference model allows an object reference to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

- The operational interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own security policies.

The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses the policy object returned to enforce the policy (as described in Section 2.2.5.3, “The Model as Seen by the Objects Implementing Security,” on page 2-62). The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

- The administrative interfaces used to set security policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so is aware of the scope of what he is administering. (Administrative interfaces are described in Section 2.2.5.2, “Administrative Model,” on page 2-58.)

Applications will often not be aware of security at all, but will still be subject to security policy, as the ORB will enforce the policies for them. Security policy is enforced automatically by the ORB both when an object invokes another and when it creates another object.

An application that knows about security can also override certain default security policy details. For example, a client can override the default quality of protection of messages to increase protection for particular messages. (Application interfaces are described in Section 2.2.5.1, “The Model as Seen by Applications,” on page 2-41.)

Note that this specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them, moving objects between them, changing the domain structure and adding, changing and removing policies applied to the domains. Such interfaces are expected to be the province of other object services and facilities.

2.2.3 Security Technology

The object security services previously described insulate the applications and ORBs from the security technology used. Security technology may be provided by existing security components. These do not have domain managers or objects. Security technology could be provided by the operating system. However, distributed, heterogeneous environments are increasingly being used, and for these, security technology is provided by a set of distributed security services. This architecture identifies a separate layer containing those components which actually implement the security services. It is envisaged that various technologies may be used to provide these and, furthermore, that a (set of) generic security interface(s) such as the GSS-API will be used to insulate the implementations of the security services from detailed knowledge of the underlying mechanisms. The range of services (and corresponding APIs) includes:

- The means of creating and handling the security information required to establish security associations, including keys.
- Message protection services providing confidentiality and integrity.

The use of standard, generic APIs for interactions with external security services not only allows interchangeability of security mechanisms, but also enables exploitation of existing, proven implementations of such mechanisms.

2.2.4 Basic Protection and Communications

2.2.4.1 Environment Domains

As described in Section 2.1.8.2, “Security Environment Domains,” on page 2-24, the way security policies are enforced can depend on the security of the environment in which the objects run. It may be possible to relax or even dispense with some security checks in the object system on interactions between objects in the same environment domain. For example, in a message protection domain where secure transport or lower layer communications is provided, encryption is not needed at the ORB level. In an identity domain, objects may share a security identity and so dispense with authenticating each other. Environment domains are implementation concepts; they do not have domain managers.

Environment domains can be exploited to optimize performance and resource usage.

2.2.4.2 Component Protection

The maintenance of integrity and confidentiality in a secure object system depends on proper segregation of the objects, which may include the segregation of security services from other components. At the lowest level of this architecture, Protection Domains, supported by a combination of hardware and software, provide a means of protecting application components from each other, as well as protecting the components that support security services. Protection Domains can be provided by various techniques, including physical, temporal, and logical separation.

The Security Architecture identifies various security services, which mediate interactions between application level components: clients and target objects. The Security Object Models show how these mechanisms can themselves be modeled and implemented in terms of additional objects. However, security services can only be effective if there is some means of ensuring that they are always invoked as required by security policies: it must be possible to guarantee, to any required level of assurance, that applications cannot bypass them. Moreover, security services themselves, like other components, must be subject to security policies.

The general approach is to establish **protection boundaries** around groups of one or more components which are said to belong to a **protection domain**. Components belonging to a protection domain are assumed to trust each other, and interactions between them need not be mediated by security services, whereas interactions across boundaries may be subject to controls. In addition, it is necessary to provide a means of establishing a trust relationship between components, allowing them to interact across protection boundaries, in a controlled way, mediated by security services.

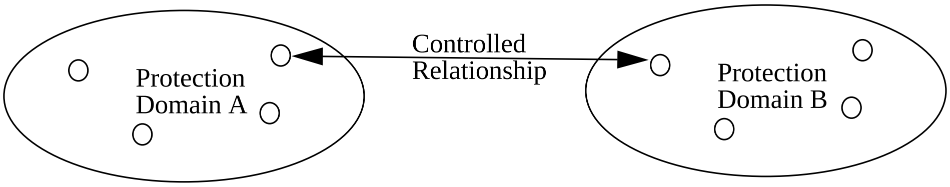


Figure 2-26 Controlled Relationship

In this architecture, the trusted components supporting security services are encapsulated by objects, as described in Section 2.2.5.3, "The Model as Seen by the Objects Implementing Security," on page 2-62. Clearly, objects that encapsulate sensitive security information must be protected to ensure that they can only be accessed in an appropriate way.

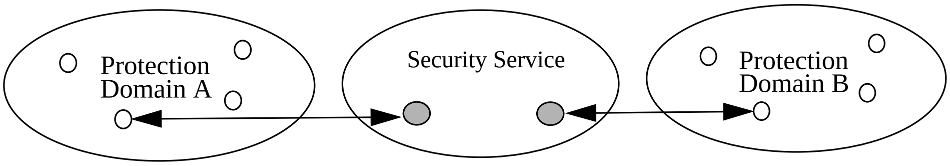


Figure 2-27 Object Encapsulation

Protection boundaries and the controlled relationships that cross those boundaries must inevitably be supported by functionality more fundamental than that of the Security Object Models, and invariably requires a combination of hardware and operating system mechanisms. Whichever way it is provided, this functionality constitutes part of the Trusted Computing Base.

Protection boundaries may be created by physical separation, interprocess boundaries, or within process access control mechanisms (e.g., multilevel "onionskin" hardware-supported access control). Less rigorous protection may be acceptable in some circumstances, and in such cases protection boundaries can be provided, for example, by using appropriate compilation tools to conceal protected interfaces and data.

The architecture is defined in a modular way so that, where necessary, it is possible for implementations to create protection boundaries between:

- Application components, which do not trust each other;
- Components supporting security services and other components;
- Components supporting security services and each other.

In addition, controlled communication across protection boundaries may be required. In such cases, it must be possible to constrain components within a protection boundary to interact with components outside the protection boundary only via controlled communications paths (it must not be possible to use alternative paths). Such communication may take many forms, ranging from explicit message passing to implicit sharing of memory.

2.2.5 Security Object Models

This section describes the objects required to provide security in a secure CORBA system from three viewpoints:

1. The model as seen by applications.
2. The model as seen by administrators and administrative applications.
3. The model as seen by the objects implementing the secure object system.

For each viewpoint, the model describes the objects and the relationships between them, and outlines the operations they support. A summary of all objects is also given.

2.2.5.1 The Model as Seen by Applications

Many applications in a secure CORBA system are unaware of security, and therefore do not call on the security interfaces. This subsection is therefore mainly relevant to those applications that are aware of and utilize security. Facilities available to such applications are:

- Finding what security features this implementation supports.
- Establishing a principal's credentials for using the system. Authenticating the principal may be necessary.
- Selecting various security attributes (particularly privileges) to affect later invocations and access decisions.
- Making a secure invocation.
- Handling security at a target object and at intermediates in a chain of objects, including use of credentials for application control of access and delegation.
- Auditing application activities.
- Non-repudiation facility -- generation and verification of evidence so that actions cannot be repudiated.
- Finding the security policies that apply to this object.

The Security Service interfaces that are available to the application writer are primarily found in the **SecurityLevel1**, **SecurityLevel2**, **NRservice**, and **SecurityAdmin** modules.

Finding Security Features

An application can find out what security features are supported by this secure object implementation. It does this by calling on the ORB to **get_service_information**. Information returned includes the security functionality level and options supported and the version of the security specification to which it conforms. It also includes security mechanisms supported (though the ORB Security Services, rather than applications, need this).

Establishing Credentials

If the principal has already been authenticated outside the object system, then **Credentials** can be obtained from **Current**.

If the principal has not been authenticated, but is only going to use public services which do not require presentation of authenticated privileges, a **Credentials** object may be created without any authenticated principal information.

If the principal has not been authenticated, but is going to use services that need him to be, then authentication is needed as shown in Figure 2-28.

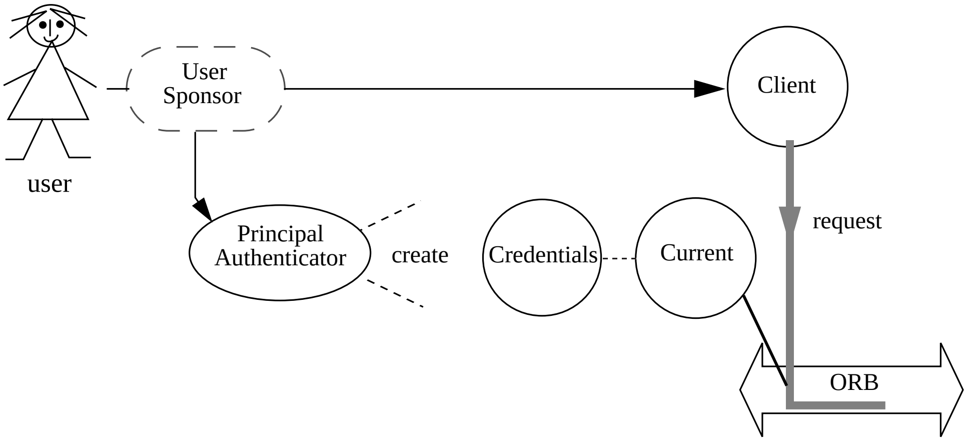


Figure 2-28 Authentication

User sponsor

The user sponsor is the code that calls the CORBA Security interfaces for user authentication. It need not be an object, and no interface to it is defined. It is described here so that the process of **Credentials** acquisition may be understood.

The user provides identity and authentication data (such as a password) to the user sponsor, and this calls on the **Principal Authenticator** object, which authenticates the principal (in this case, the user) and obtains **Credentials** for it containing authenticated identity and privileges.

The user sponsor represents the entry point for the user into the secure system. It may have been activated, and have authenticated the user, before any client application is loaded. This allows unmodified, security-unaware client applications to have **Credentials** established transparently, prior to making invocations.

There is no concept of a target object sponsor.

Principal Authenticator

The **Principal Authenticator** object is the application-visible object responsible for the creation of **Credentials** for a given principal. This is achieved in one of two ways. If the principal is to be authenticated within the object system, the user sponsor invokes the **authenticate** operation of the **Principal Authenticator** object (and **continue_authentication** if needed for multiexchange authentication dialogues).

Credentials

A **Credentials** object holds the security attributes of a principal. These security attributes include its authenticated (or unauthenticated) identities and privileges and information for establishing security associations. It provides operations to obtain and set security attributes of the principal it represents.

There may be credentials for more than one principal, for example, the initiating principal who requested some action and the principal for the current active object. **Credentials** are used on invocations and for non-repudiation.

There is an **is_valid** operation to check if the credentials are valid and a **refresh** operation to refresh the credentials if possible.

Current

The **Current** object represents the current execution context at both client (both for object or non-object clients) and target objects. In a secure environment, the interfaces **SecurityLevel1::Current** which is derived from **CORBA::Current** and **SecurityLevel2::Current** which is derived from **SecurityLevel1::Current**, give access to security information associated with the execution context. **Current** gives access to the **Credentials** associated with the execution environment. Object invocations use **Credentials** in **Current**, unless they have been overridden, by a security aware client, in the specific object reference being used for the invocation. If a user sponsor is used, it should set the user's credentials for subsequent invocations in **Current**. This may also be done as the result of initializing the ORB when the user has been authenticated outside the object system. This allows a security-unaware application to utilize the credentials without having to perform any explicit operation on them.

At target and intermediate objects, other **Credentials** are also available via **Current**.

Handling Multiple Credentials

An application object may use different **Credentials** with different security characteristics for different activities.

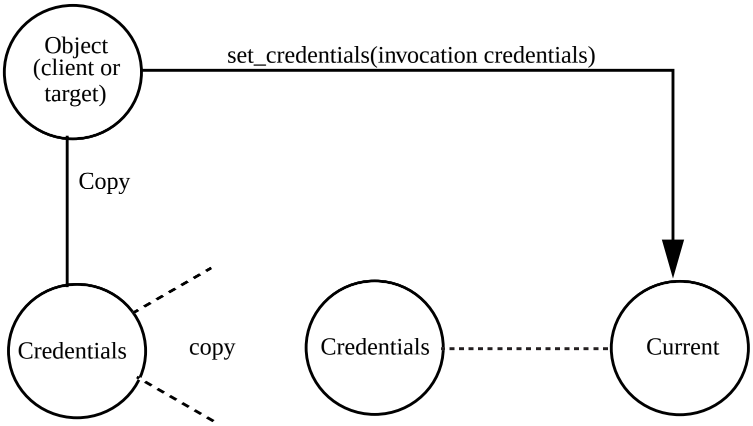

Object (client or target)
set_credentials(invocation credentials)
Copy
Credentials
copy
Credentials
Current

Figure 2-29 Multiple Credentials

The **Credentials::copy** operation can be used to make a copy of the **Credentials** object. The new **Credentials** object (i.e., the copy) can then be modified as necessary, using its interface, before it is used in an invocation.

When all required changes have been made, the **Current::set_credentials** operation can be used to specify a different **Credentials** object as the default for subsequent invocations.

At any stage, a client or target object can find the default credentials for subsequent invocations by calling **Current::get_credentials**, asking for the invocation credentials. These default credentials will be used in all invocations using object references in which the invocation credentials have not been overridden.

Selecting Security Attributes

A client may require different security for different purposes, for example, to enforce a least privilege policy and so specify that limited privileges should be used when calling particular objects, or collections of objects, and restrict the scope to which these privileges are propagated. A client may also want to protect conversations with different targets differently.

There are two ways to change security attributes for a principal:

1. Setting attributes on the credentials for that principal. If attributes are set on the credentials, these apply to subsequent object invocations using those credentials. It can therefore apply to invocations of many target objects.

2. Overriding attributes on the target object reference. Attributes thus set apply to subsequent invocations, which this client makes using this reference.

In both cases, the change applies immediately to further object invocations associated with these credentials or this object reference.

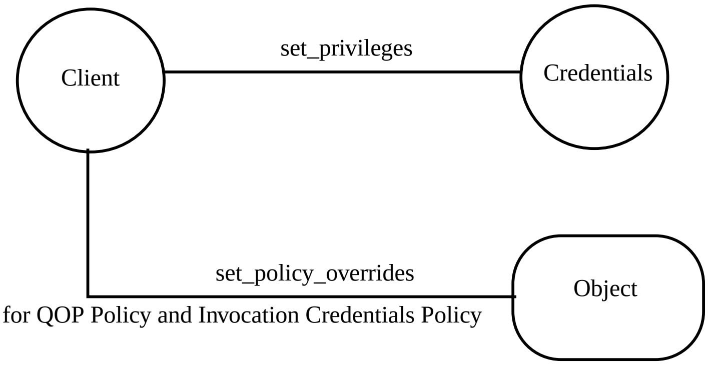


Figure 2-30 Changing Security Attributes

A wider range of attributes can be set on the credentials than on a specific object reference. Operations available include:

- **set_privileges** to set privileges in the credentials. The system will reject an attempt to set privileges if the calling principal is not entitled to one or more of the requested privileges. There may be additional restrictions on which privileges may be claimed if the caller is an intermediate in a delegated call chain attempting to set privileges on delegated **Credentials**.

Setting any of these attributes may result in a new security association being needed between this client and target.

Note – This specification does not contain an operation to restrict when and where these privileges can be used in target objects or delegated, though this may be specified in the future (see Appendix F, Section F.12, “Target Control of Message Protection,” on page F-5).

A client may want to use different privileges or controls when invoking different targets. It can do this by obtaining a new object reference using the **set_policy_overrides** specifying the invocation credentials policy to be used with that target, and then use the object reference thus obtained to carry out the invocation.

A client may want to specify that a particular quality of protection applies only to selected invocations of a target object. For example, it may want confidentiality of selected messages. The client can do this by using **set_policy_overrides**, specifying a **QOP Policy** on the new object reference. It can continue to use the original object reference for those invocations where confidentiality is no longer required.

The **set_policy_overrides** operation returns a new object reference to the same target object as the one on which this operation is invoked. This new reference has the policy overrides set in it. Any invocations through this new reference will use the overrides set in the reference. The creation of this newly annotated object reference has no effect on the target object.

Equivalent **get_** operations are also provided to permit an application to determine the security specific options currently requested, for example **get_attributes** (privileges, and other attributes such as audit id).

The *security features*, *invocation credentials*, *qop*, and *mechanism* related policies that are in effect on a given object reference can be obtained by using the **get_policy** operation asking for the appropriate type of policy object.

Making a Secure Invocation

A secure invocation is made in the same way as any other object invocation, but the actual invocation is mediated by the ORB Security Services, invisibly to the application, which enforce the security requirements, both in terms of policy and application preference. The following diagram shows an application making the invocation, and the ORB Security Services utilizing the security information in **Current**, and hence the **Credentials** there.

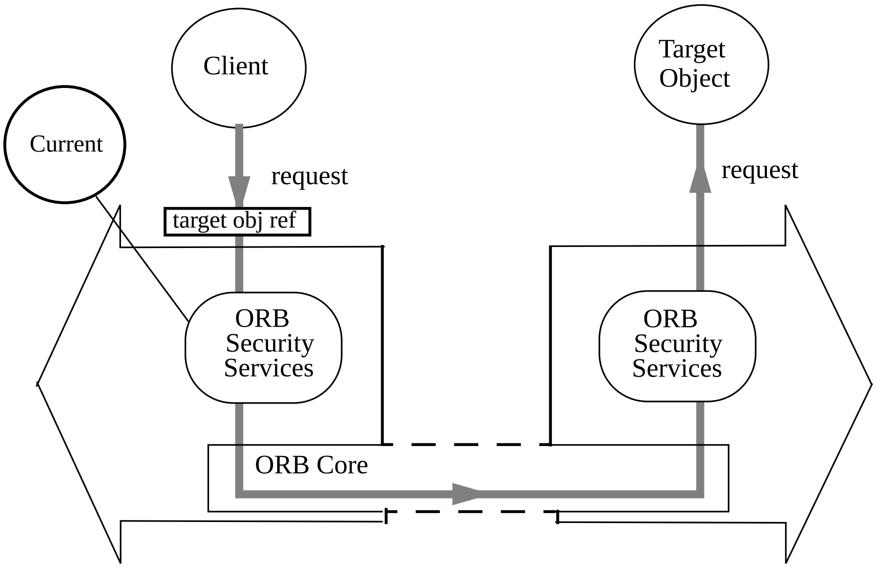


Figure 2-31 Making a Secure Invocation

Note – For any given invocation, it is target and client security policy that determines which (if any) ORB Security Services mediate that invocation. If the policy for a given invocation requires no security, then no services will be used. Similarly, if only access control is required, then only the ORB Security Service responsible for the provision of access control will be invoked.

Security at the Target

At the target, as at the client, the **Current** object is the representative of the local execution context within which the target object's code is executing. The **Current** object can be used by the target object, or by ORB and Object Service code in the target object's execution context, to obtain security information about an incoming security association and the principal on whose behalf the invocation was made.

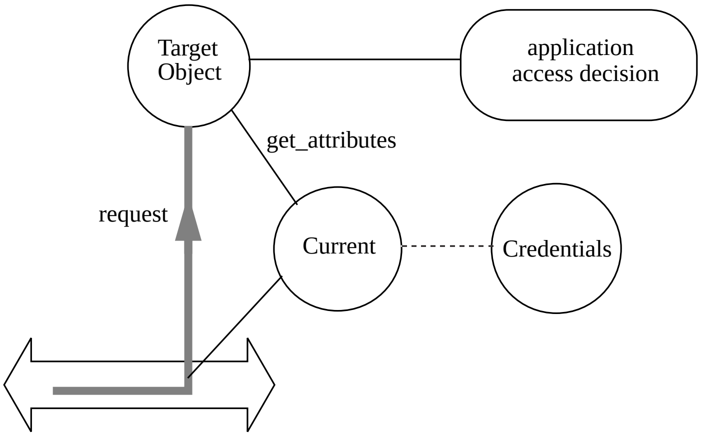


Figure 2-32 Target Object Security

A security-aware target application may obtain information about the attributes of the principal responsible for the request by invoking the **Current::get_attributes** operation. The target normally uses **get_attributes** to obtain the privilege attributes it needs to make its own access decisions.

When **Current::get_attributes** is invoked from the target object it returns the attributes from the incoming **Credentials** from the client. When **Current::get_attributes** is invoked by a client the attributes from the **Credentials** of the user (e.g., the one that was created by the **PrincipalAuthenticator**) is returned. Invoking **Credentials::get_attribute** always returns the attributes contained in that **Credentials** object.

Intermediate Objects in a Chain of Objects

When a client calls a target object to perform some operation, this target object often calls another object to perform some function, which calls another object and so on. Each intermediate object in such a chain acts first as a target, and then as a client, as shown in Figure 2-33 on page 2-48.

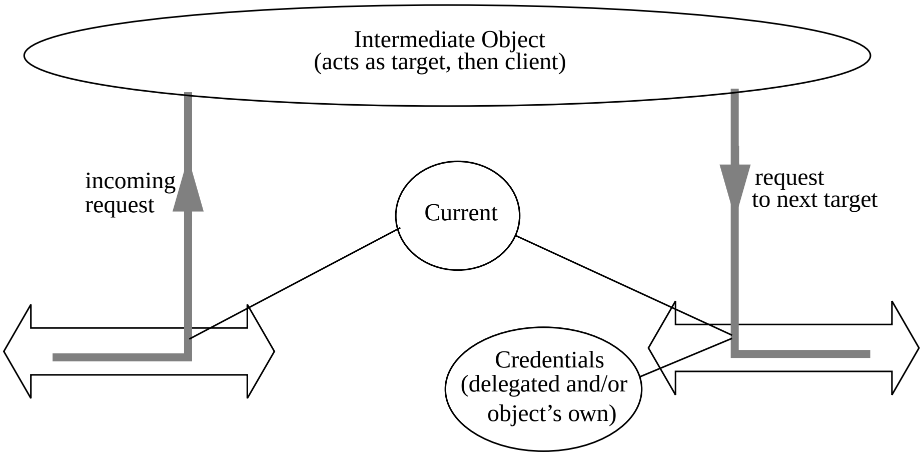


Figure 2-33 Security-unaware Intermediate Object

For a security-unaware intermediate object, **Current** has a reference to the security context established with the incoming client. When this intermediate object invokes another target, either the delegated credentials from the client or the credentials for the intermediate object's principal (or both) become the current ones for the invocation. The security policy for this intermediate object governs which credentials to use, and the ORB Security Services enforce the policy, passing the required credentials to the target, subject to any delegation constraints. The intermediate object's principal will be authenticated, if needed, by the ORB Security Services.

A security-aware intermediate object can:

- Use the privileges of any delegated credentials for access control.
- Decide which credentials to use when invoking further targets.
- Restrict the privileges available via these credentials to further clients (where security technology permits).

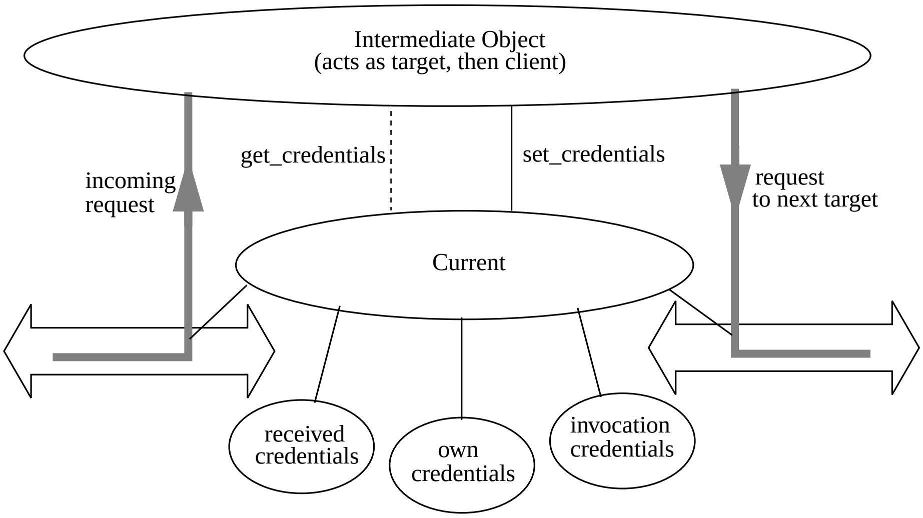


Figure 2-34 Security-aware Intermediate Object

After a chain of object calls, the target can call **Current::get_attributes** as previously described. Note that this call always obtains the privilege and other attributes associated with the first of the received credentials.

The target can use the **received_credentials** attribute of **Current** to get the incoming credentials. After a composite delegation (see Section 2.1.6, "Delegation," on page 2-13), the credentials are of the initiator and immediate invoker. After traced delegation, credentials for all intermediates in the chain will be present (as well as the initiator). If a target object receives a request which includes credentials for more than one principal, it may choose which privileges to use for access control and which credentials to delegate, subject to policy.

An intermediate object may wish to make a copy of the incoming credentials, modify and then delegate them, though not all implementations will support this modification. In this case, it must acquire a reference to the incoming credentials (using the **received_credentials** attribute of **Current**), and then use **Credentials::set_privileges** to modify them. Finally it can call **Current::set_credentials** to make the received credentials the default ones for subsequent invocations. When the **received_credentials** are passed to **set_credentials**, whether it is a delegation or not needs to be specified to the **set_credentials** operation, and it takes appropriate action.

If the intermediate object wishes to change the association security defaults (for example, the quality of protection) for subsequent invocations to a specific target object, it can do so by using the **Object::set_policy_overrides** operation to create a copy of the object reference to the target with the required QOP set as override in the object reference thus obtained. The overridden QOP will apply to subsequent invocations through this new reference.

The intermediate object may be a principal and wish to use its own identity and some specific privileges in further invocations, rather than delegating the ones received. In this case, it can call **authenticate** operation of the **PrincipalAuthenticator** to obtain the appropriate credential, and then call **Credentials::set_privileges** to establish the appropriate rights. After doing this, it can use **Current::set_credentials** to establish its credential as the default for future invocations.

If the intermediate does not have its own individual **Credential** object (for example, as it does not have an individual security name) but instead shares credentials with other objects, it can us the **own_credentials** attribute of **Current** to get a copy of the **Credentials** (which will have been set up automatically). It can then do a **Credentials::copy** and then a **Credentials::set_privileges**, etc. on these, as appropriate and then use it to obtain a new object reference for the object it intends to invoke, with invocation credentials policy overridden using the **Credentials** constructed above.

If it wants to use composite delegation with a modified version of its own credentials, it should call **Current::set_credentials** (specifying its own credentials) and the required delegation mode before making the invocation. Note that this will not modify the credentials shared with other objects.

Security Mechanisms

Applications are normally aware of the security mechanism used to secure invocations. The secure object system is aware of the mechanisms available to both client and target object and can choose an acceptable mechanism. However, some security-sophisticated applications may need to know about, or even control the choice of mechanisms. They can get information on the currently in effect mechanism policy by using the **get_policy** operation of the object reference. They can do invocations using a different mechanism from the default by using **set_policy_overrides** operation of the object reference to obtain a new object reference with the desired mechanism policy set as override in it and use it for invocations that need the new mechanism.

Application Access Policies

Applications can enforce their own access policies. No standard application access policy is defined, as different applications are likely to want different criteria for deciding whether access is permitted. For example, an application may want to take into account data values such as the amount of money involved in a funds transfer.

However, it is recommended that the application use an access decision object similar to the one used for the invocation access policy. This is to isolate the application from details of the policy. Therefore, the application should decide if access is needed as shown in Figure 2-35 on page 2-51.

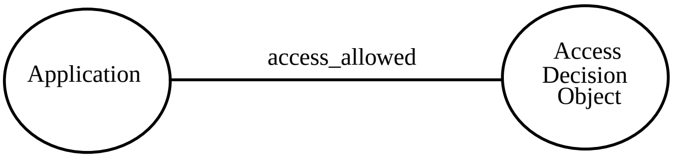


Figure 2-35 access_allowed Application

The application can specify the privileges of the initiating principal and a variety of authorization data, which could include the function being performed, and the data it is being performed on.

An application access policy can be used to supplement the standard invocation access policy with an application-defined policy. Such a policy might, for example, take into account the parameters to the request. In this case, the authorization data passed to the application-defined policy would be likely to include the request's operation, parameters, and target object.

The application access policy could be associated with the domain, and managed using the domain structure as for other policies (see Section 2.2.5.2, "Administrative Model," on page 2-58). In this case, the application obtains the **Access Policy** object as shown in Figure 2-36.

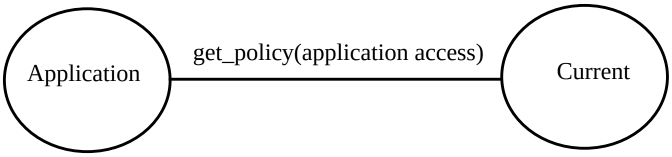


Figure 2-36 get_policy Application

However, the application could choose to manage its access policy differently.

Auditing Application Activities

Applications can enforce their own audit policies, auditing their own activities. Audit policies specify the selection criteria for deciding whether to audit events.

As for application access policies, application audit policies can be associated with domains and managed via the domain structure. No standard application level audit policy is specified, as different applications may want to use different selectors in deciding which events to audit. Application events are generally not related to object invocations. Applications can provide their own audit policies, which use different criteria. The most common selectors for these audit policies to use are the event type and its success or failure, the **audit_id** and the time. (Management of such policies can generally be done using the interfaces for audit policy administration defined in Section 2.4.5, "Audit Policies," on page 2-131, by specifying new selectors, appropriate to the application concerned.)

Whether or not the application uses an audit policy, it uses an **Audit Channel** object to write the audit records. One Audit Channel object is created at ORB initialization time, and this is used for all system auditing. Applications can use different audit channels.

The way an Audit Channel object handles the audit records is not visible to the caller. It may filter them, route them to appropriate audit trails, or cause event alarms. Different Audit Channel objects may be used to send audit records to different audit trails.

Applications and system components both invoke the **audit_write** operation to send audit records to the audit trail.

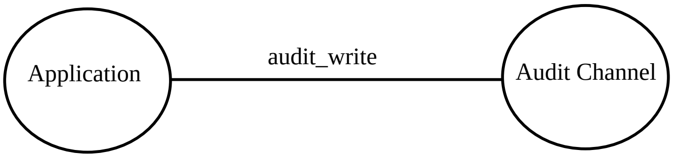


Figure 2-37 audit_write Application

If an application is using an audit policy administered via domains, it uses an **Audit Decision** object (see Section 2.3.8, "Security Audit," on page 2-100) to decide whether to audit an event. It can find the appropriate **Audit Decision** object using the **audit_decision** attribute of **Current** as follows.

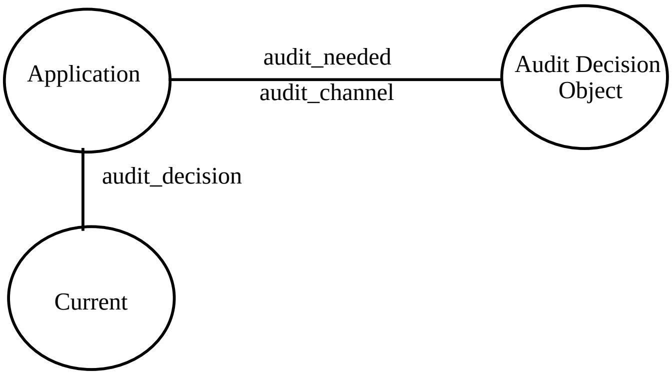


Figure 2-38 Audit Decision Object

The application invokes the **audit_needed** operation of the **Audit Decision** object, passing the values required to decide whether auditing is needed. (This set of selectors could include, for example, the type of event, its success or failure, the identity of the caller, the time, etc. See administration of audit policies in Section 2.3.8, "Security Audit," on page 2-100.)

The audit channel to be used in conjunction with an audit policy object can be identified to the audit policy object with an audit channel id. The **Audit Decision** object uses this **Audit Channel Id** to gain access to the corresponding **Audit Channel** and return it to the user. Thus the application can use an **Audit Channel** associated with the application (and these can link into the system audit services). If so, the application uses the **audit_channel** attribute of the Audit Decision object to find the **Audit Channel** object to use. However, applications can create their own **Audit Channels** with the help of the underlying audit service, and register their **Audit Channel Ids** with the appropriate **Audit Policy** object. The association between the **Audit Channel Id** and the audit channel is maintained by the underlying audit service, which is not specified in this chapter.

Finding What Security Policies Apply

An application may want to find out what policies the system is enforcing on its behalf. For example, it may want to know the default quality of protection to be used by default for messages or for non-repudiation evidence.

To do this, it can call **Current::get_policy**, and then the appropriate **get_** operation of the policy object obtained as defined in Section 2.4, "Administrator's Interfaces," on page 2-116 (if permitted).

Non-repudiation

The non-repudiation services in this specification provide generation of evidence of actions and later verification of this evidence, to prove that the action has occurred. There is often data associated with the action, so the service needs to provide evidence of the data used, as well as the type of action.

These core facilities can be used to build a range of non-repudiation services. It is envisioned that *delivery services* will be implemented to deliver this evidence to where it is needed and *evidence stores* will be built for use by adjudicators. As different services may have different requirements for these, interfaces for them are not included in this specification.

Non-repudiation Credentials and Policies

Non-repudiation operations are performed on **NRCredentials**. As for any other **Credentials** object, these hold the identity and attributes of a principal. However, in this case, the attributes include whatever is needed for identifying the user for generating and checking evidence. For example, it might include the principal's key (or provide access to it) as needed to sign the evidence.

NRCredentials are available via the **Current** object as for other **Credentials** objects, and support the operations defined for credentials previously described. The credentials to be used for non-repudiation can be specified using the **set_credentials** operation on **Current** with a type of **NRCredentials**.

An application can set security attributes related to non-repudiation using the **NRCredentials::set_NR_features** operation.

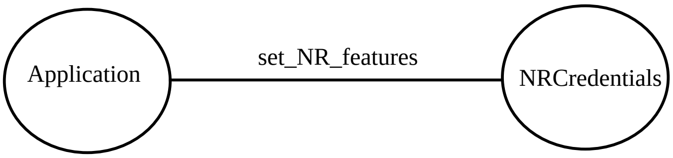


Figure 2-39 set_NR_features Operation

The **set_NR_features** can be used to specify, for example, the quality of protection and the mechanism to be used when generating evidence using these credentials.

By default, the features are those associated with the non-repudiation policy obtained by invoking **Current::get_policy** specifying **Security::SecNonRepudiation**. However, non-repudiation policies may come from other sources. For example, the policy to be used when generating evidence for a particular recipient may be supplied by that recipient.

There is an **NRCredentials::get_NR_features** operation equivalent to **set_NR_features**.

Evidence generation and verification operations are also performed on **NRCredentials** objects. These are described next.

Using Non-Repudiation Services

An application can generate evidence associated with an action so that it cannot be repudiated at a later date. All evidence and related information is carried in non-repudiation tokens. (The details of these are mechanism specific.)

The application decides that it wishes to generate some proof of an action and calls the **generate_token** operation of an **NRCredentials** object.

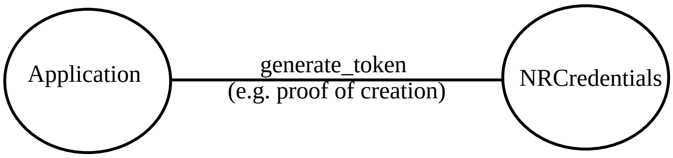


Figure 2-40 generate_token Operation

This evidence is created in the form of a non-repudiation token rendered unforgeable. Generation of the token uses the initiating principal's security attributes in the **NRCredentials** (normally a private key), for example, to sign the evidence.

Depending on the underlying cryptographic techniques used, the evidence is generated as:

- A secure envelope of data based on symmetric cryptographic algorithms requiring what is termed to be a trusted third party as the evidence generating authority.
- A digital signature of data based on asymmetric cryptographic algorithms which is assured by public key certificates, issued by a Certification Authority.

Depending on the non-repudiation policy in effect for a specific application and the legal environment, additional information (such as certificates or a counter digital signature from a Time Stamping Authority) maybe required to complete the non-repudiation information. A time reference is always provided with a non-repudiation token. A Notary service may be required to provide assurance about the properties of the data.

Complete Evidence

Non-repudiation evidence may have to be verified long after it is generated. While the information necessary to verify the evidence (e.g., the public key of the signer of the evidence, the public key of the trusted time service used to countersign the evidence, the details of the policy under which the evidence was generated, etc.) will ordinarily be easily accessible at the time the evidence is generated, that information may be difficult or impossible to assemble a long time afterward.

The CORBA Non-repudiation Service provides facilities for incorporating all information necessary for the verification of a piece of non-repudiation evidence inside the evidence token itself. A token including both non-repudiation evidence and all information necessary to verify that evidence is said to contain “complete” evidence.

There may be policy-related limitations on the time periods during which complete evidence may be formed. For example, Non-repudiation policy may permit addition of the signer’s public key to the evidence only after expiration of the interval, during which the signer may permissibly declare that key to have been compromised. Similarly, the policy may require application of the Trusted Time Service countersignature within a specified interval after application of the signer’s signature.

To facilitate the generation of complete evidence, the information returned from the calls which verify evidence and request formation of complete evidence, includes two indicators (**complete_evidence_before** and **complete_evidence_after**) indicating the earliest time at which complete evidence may usefully be requested and the latest time at which complete evidence can successfully be formed.

A call to **verify_evidence** before complete evidence can be formed may result in a response declaring the evidence to be “conditionally valid.” This means that the evidence is not invalid at the current time, but a future event (e.g., the signer declaring his key compromised) might cause the evidence to be invalid when complete.

Figure 2-41 on page 2-56 illustrates the policy considerations relating to generation of complete evidence, and the sequence of actions involved in generating and using complete evidence.

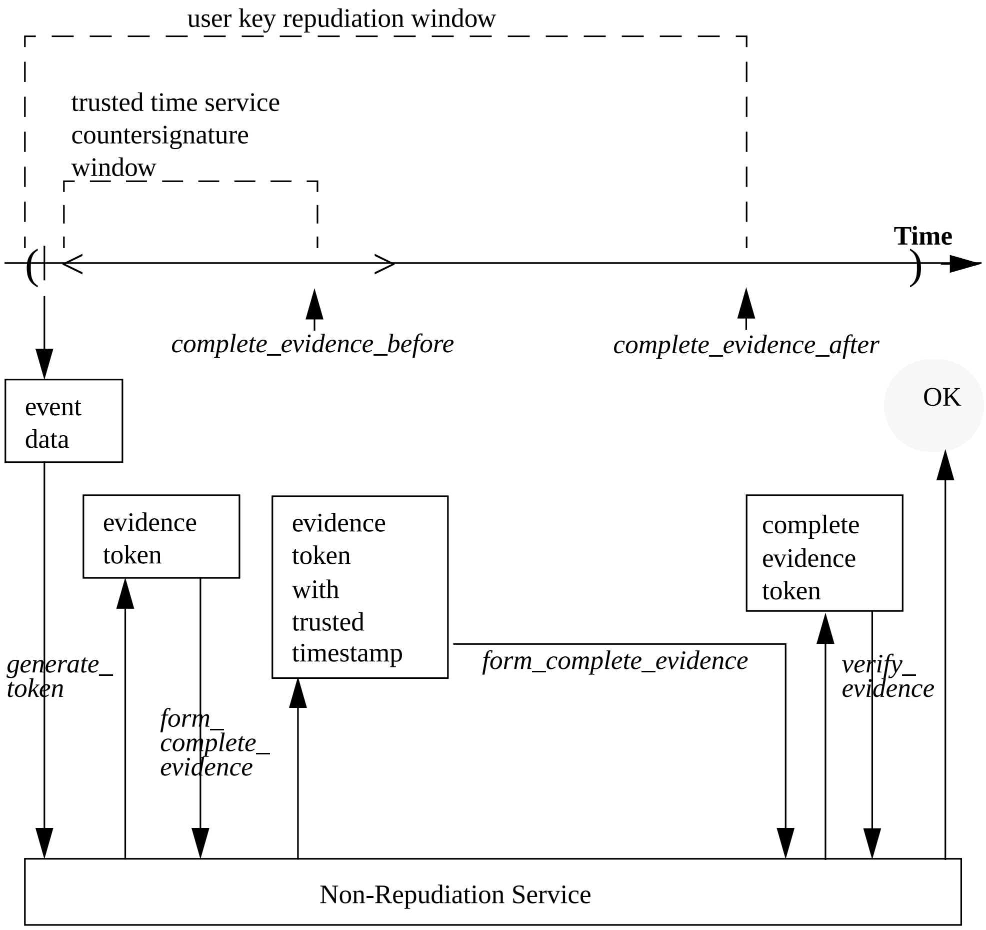


Figure 2-41 Non-repudiation Service

An application may receive a token and need to know what sort of token it is. This is done using **get_token_details**. When the token contains evidence, **get_token_details** can be used to extract details such as the non-repudiation policy, the evidence type, the originator's name, and the date and time of generation. These details can be used to select the appropriate non-repudiation policy and other features (using **set_NR_features**), as necessary for verifying the evidence. When the token contains a request to send back evidence to one or more recipients, then if appropriate, evidence can be generated.

An application verifies the evidence using the **verify_evidence** operation.

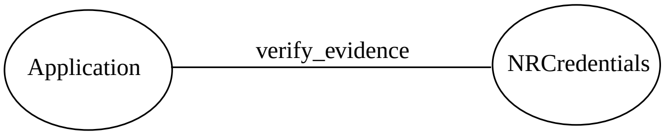


Figure 2-42 verify_evidence operation

Verification of non-repudiation tokens uses information associated with the Non-repudiation Policy applicable to the non-repudiation token and security information about the recipient who is verifying the evidence (normally the public key from a Certification Authority and a set of trust relationships between Certification Authorities).

Using Non-Repudiation for Receipt of Messages

An application receiving a message with proof of origin may handle it as shown in Figure 2-43.

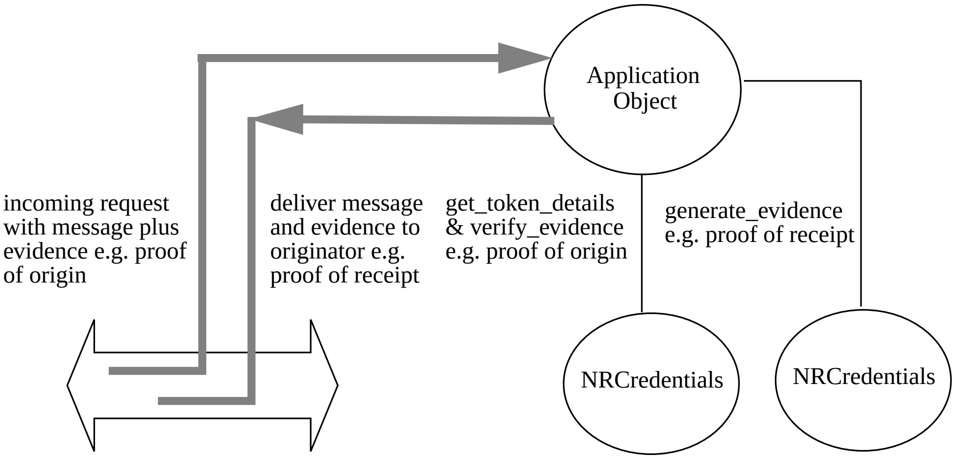

Application Object
incoming request with message plus evidence e.g. proof of origin
deliver message and evidence to originator e.g. proof of receipt
get_token_details & verify_evidence e.g. proof of origin
generate_evidence e.g. proof of receipt
NRCredentials
NRCredentials

Figure 2-43 Proof of Origin Message

- The application receives the incoming message with a non-repudiation token that has been generated by the originator.
- The application now wishes to know the type of token that it has received. It does this by calling the **NRCredentials::get_token_details** operation. The token may be:
  - A request that evidence be sent back (such as an acknowledge of receipt)
  - Evidence of an action (such as a proof of creation)
  - Both evidence and a request for further evidence.
- The application's next action depends on which of the three cases applies.
  - In the first case, the application verifies that it is appropriate to generate the requested evidence and, if so, generates that evidence using **NRCredentials::generate_token**.

- In the second case, the application retrieves the data associated with the evidence if it is outside the token, and verifies the evidence using **NRCredentials::verify_evidence**, presenting the token alone or the concatenation of the token and the data.
- In the last case, the application verifies the received evidence by first calling **NRCredentials::verify_evidence**, and then generating evidence if appropriate, as in the first case.

- If the application receives a token that contains valid evidence, and wishes to store it for later use, it needs to make sure that it holds all the necessary information. It may need to call **NRCredentials::form_complete_evidence** in order to get the complete evidence needed when this could not be provided using the verify operation.
- When the application has generated evidence as the result of a request from the originator of the message, the application must send it to the various recipients as indicated in the NR token received.

Using Non-repudiation Services for Adjudication

Adjudication applications use the **NRCredentials::verify_evidence** operation, which must return complete evidence to settle disputes.

2.2.5.2 Administrative Model

The administrative model described here is concerned with administering security policies.

- Administration of security environment domains and security technology domains may be implementation specific, so it is not covered here. This means administrating security technology specific objects is out of the scope of this specification.
- Explicit management of nonsecurity aspects of domains is not covered.

Administrative activities covered here are:

- Creating objects in a secure environment subject to the security policies
- Finding the domain managers that apply to this object.
- Finding the policies for which these domain managers are responsible.
- Setting security policy details for these policy objects.
- Specifying which rights give access to which operations in support of access policies.

The model used here is not specific to security, though the specific policies described are security policies.

Security Policies

Security policies may affect the security enforced:

- By applications. In general, enforcing policy within applications is an application concern, so it is not covered by this specification. However, where the application uses underlying security services, it will be subject to their policies.
- By the ORB Security Services during object invocation (the main focus of this specification).
- In other security object services, particularly authentication and audit.
- In any underlying security services. (In general, this is not covered by this specification, as these security services are often security technology specific.)

This specification defines the following security policy types:

- **Invocation access policy**
  The object that implements the access control policy for invocations of objects in this domain.
- **Invocation audit policy**
  This controls which types of events during object invocation are audited, and the criteria controlling auditing of these events.
- **Secure invocation policy**
  This specifies security policies associated with security associations and message protection. For example, it specifies:
  - Whether mutual trust between client and target is needed (i.e., mutual authentication if the communications path between them is not trusted).
  - Quality of protection of messages (integrity and confidentiality).

There may be separate invocation policies for applications acting as client and those acting as target objects in this domain. This applies to access, audit, and secure invocation policies. There may also be separate policies for different types of objects in the domain.

- **Invocation delegation policy**
  This controls whether objects of the specified type in this domain, when acting as an intermediate in a chain, by default delegate the received credentials, use their own credentials, or pass both.
- **Application access policy**
  This policy type can be used by applications to control whether application functions are permitted. Unlike invocation policies, it does not have to be managed via the domain structure, but may be managed by the application itself.
- **Application audit policy**
  This policy type can be used by applications to control which types of application events should be audited under what circumstances.
- **Non-repudiation policy**
  Where non-repudiation is supported, a non-repudiation policy has the rules for generation and verification of evidence.
- **Construction policy**
  This controls whether a new domain is created when an object of a specific type is created.

Domains at Object Creation

Any object that is accessible through an ORB must have an object reference created for it. This is often done as a part of the procedure for creating the object by a factory object. When a new object reference is created in a secure environment, the ORB implicitly associates the object reference, and hence the associated object, with the following elements forming its environment.

- One or more *Security Policy Domains*, defining all the policies to which the object is subject.
- The *Security Technology Domains*, characterizing the particular variants of security mechanisms available in the ORB.
- Particular *Security Environment Domains* where relevant.

The application code involved in the creation of an object, and its reference may not need to be aware of security to protect the objects it creates, if the details are encapsulated in a Factory object. Automatically making an object reference and hence the associated object a member of policy domains on creation ensures that mandatory controls of enclosing domains are not bypassed.

The ORB will establish these associations when the creator of the object calls **PortableServer::POA::create reference** or **PortableServer::POA::create_referece_with_id** (see the Portable Object Adaptor chapter of the *Common Object Request Broker: Architecture and Specification*) or an equivalent. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object reference is created, a new domain is also needed. For example, in a banking system, there may be a domain for each bank branch, which provides policies for bank accounts at that branch. Therefore when a bank branch is created, a new domain is needed. As for a newly created object's domain membership, if the application code creating the object and the object reference to it is to be unaware of security, the domain manager must be created transparently to the application. A construction policy specifies whether new objects reference of this type in this domain require a new domain.

This construction policy is enforced at the same time as the domain membership (i.e., by **POA::create_reference*** or equivalent). For details, see the Portable Object Adaptor chapter of the *Common Object Request Broker: Architecture and Specification*.

Other Domain and Policy Administration

Once an object reference has been created as a member of a policy domain, it may be moved to other domains using the appropriate domain management facilities (not specified in this chapter).

Once a domain manager has been created, new security policy objects can be associated with it using the appropriate domain management facilities. These security policy objects are administered as defined in this specification.

The following diagram shows the operations needed by an administrative application to manage security policies.

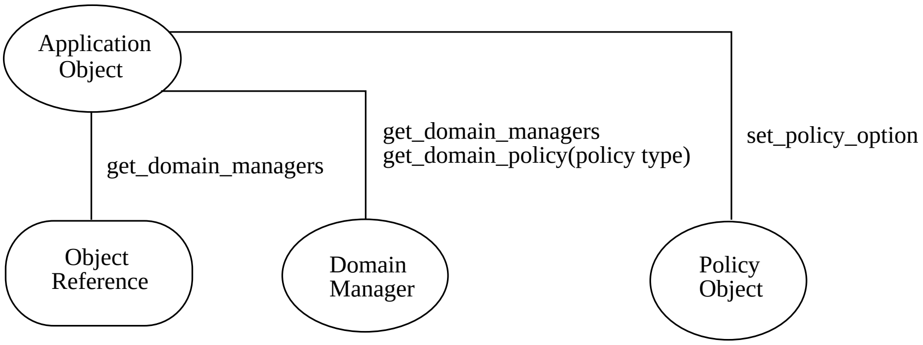


Figure 2-44 Managing Security Policies

Finding Domain Managers

An application can invoke the **get_domain_managers** operation on an object reference to obtain a list of the immediately enclosing domain managers for that object (i.e., the object associated with the object reference). If these do not have the type of policy required, a call can be made to **get_domain_managers** on one of these domain managers to find its immediately enclosing domains.

Finding the Policies

Having found a domain manager, the administrative application can now find the security policies associated with that domain by calling **get_domain_policy** on the domain manager specifying the type of policy it wants (e.g., client secure invocation policy, application audit policy). This returns the **Policy** object needed to administer the policy associated with this domain. Each **Policy** object supports the operations required to administer that policy.

In this specification, no facilities are provided to specify the rules for combining policies for overlapping domains, though some implementations may include default rules for this. (Definition of such rules is a potential candidate for future security specifications. See Appendix F, “Facilities Not in This Specification.”)

If the policy that applies to the domain manager’s own interface is required (rather than the one for the objects in the domain), then **get_policy** (rather than **get_domain_policy**) is used.

Setting Security Policy Details

Having found the required security Policy object, the application uses its interface to set the policy.

The operations available through the interface depend on the type of policy. For example, the delegation policy only requires a delegation mode to be set to specify delegation mode used when the object acts as an intermediate in a chain of object invocations, whereas an access policy will need to have an operation that makes it possible to specify who can access the objects.

Administrative interfaces are defined in Section 2.4, “Administrator’s Interfaces,” on page 2-116, for the standard policy types, which all ORBs supporting security functionality Level 2 support.

Different administration may be needed if standard policies are replaced by different policies. A supplier providing another policy may therefore have to specify its administrative interfaces.

Specifying Use of Rights for Operation Access

The access policy is used to decide whether a user with specified privileges has specified *rights*. A specific right may permit access to exactly one operation. More often, the right permits access to a set of operations.

A **RequiredRights** object specifies which rights are required to use which operations of an interface. The administrator can **set_required_rights** on this object.

2.2.5.3 *The Model as Seen by the Objects Implementing Security*

Security is provided for security-unaware applications by implementation level security objects, which are not directly accessible to applications. These same implementation objects are also used to support the application-visible security objects and interfaces described in Section 2.2.5.1, “The Model as Seen by Applications,” on page 2-41 and Section 2.2.5.2, “Administrative Model,” on page 2-58.

There are two places where security is provided for applications, which are unaware of security. These are:

1. On object invocation when invocation time policies are automatically enforced.
2. On object creation, when an object automatically becomes a member of a domain, and therefore subject to the domain’s policies.

Implementor’s View of Secure Invocations

Figure 2-45 on page 2-63 shows the implementation objects and services used to support secure invocations.

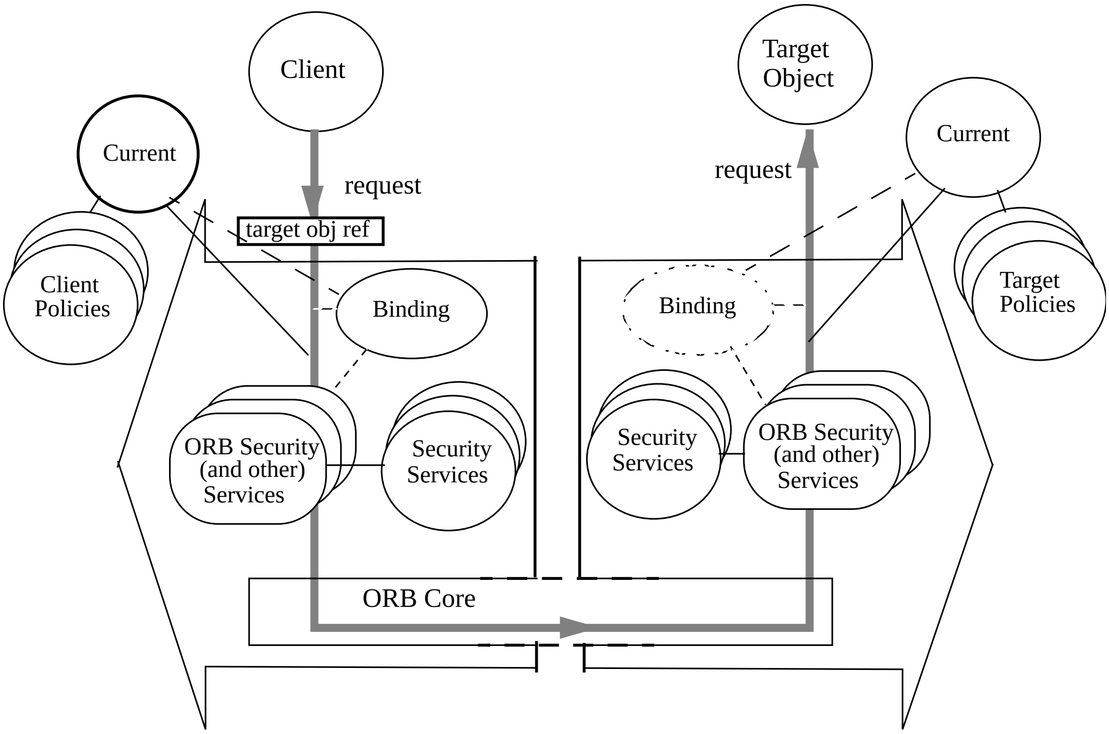


Figure 2-45 Securing Invocations

ORB Security Services

ORB Security Services are interposed in the path between the client and target object to handle the security of the object invocation. They may be interspersed with other ORB services, though where message protection is used, this will be the last ORB service at the client side, as the request cannot be changed after this.

The ORB services use the policy objects to find which policies to apply to the client and target object, and hence the invocation. The ORB and ORB Services establish the binding between client and target object as defined in ORB Services, under Section 2.2.2, “Structural Model,” on page 2-32. The ORB Security Services call on the security services to provide the required security.

Security Policy

At the client, the security policies associated with it are accessed by the ORB Security Services using the **Current::get_policy** operation specifying the type of policy required. At the client, the invocation policies that will be used for a specific invocation through a specific object reference can be inspected using the **get_policy** operation on that object reference. At the target, **Current::get_policy** is used in a similar way to obtain the policy associated with the target object.

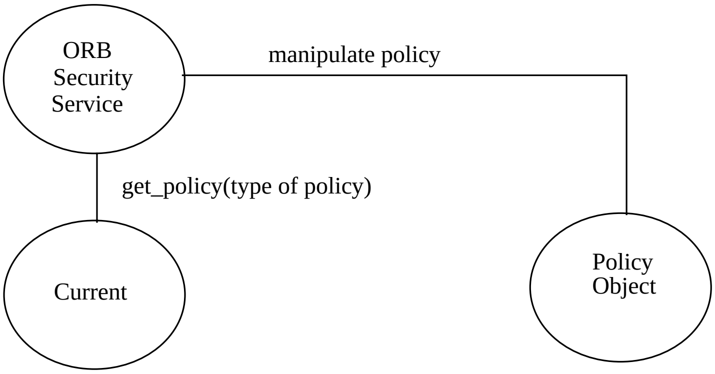


Figure 2-46 get_policy Operation

Once the policy object has been obtained, the ORB Service uses it to enforce policy. The operations used to enforce the policy depend on the type of policy. In some cases, such as secure invocation or delegation, the ORB Service invokes a **get_** operation of the appropriate **Policy** object (e.g., **SecureInvocationPolicy::get_association_options**, **DelegationPolicy::get_delegation_mode**) specifying the particular policy options required (e.g., whether confidentiality is required, and the delegation mode, respectively). It then uses this information to enforce the policy, for example, pass the required policy options to the **Vault** to enforce.

Decision objects corresponding to certain policy objects include rules, which enforce the policy. For example, an access decision object corresponding to the access policy object has the **access_allowed** operation, which responds with a yes or no.

Specific ORB Security Services and Replaceable Security Services

The specific ORB Security Services and security services included in the CORBA security object model are shown in Figure 2-47 on page 2-65.

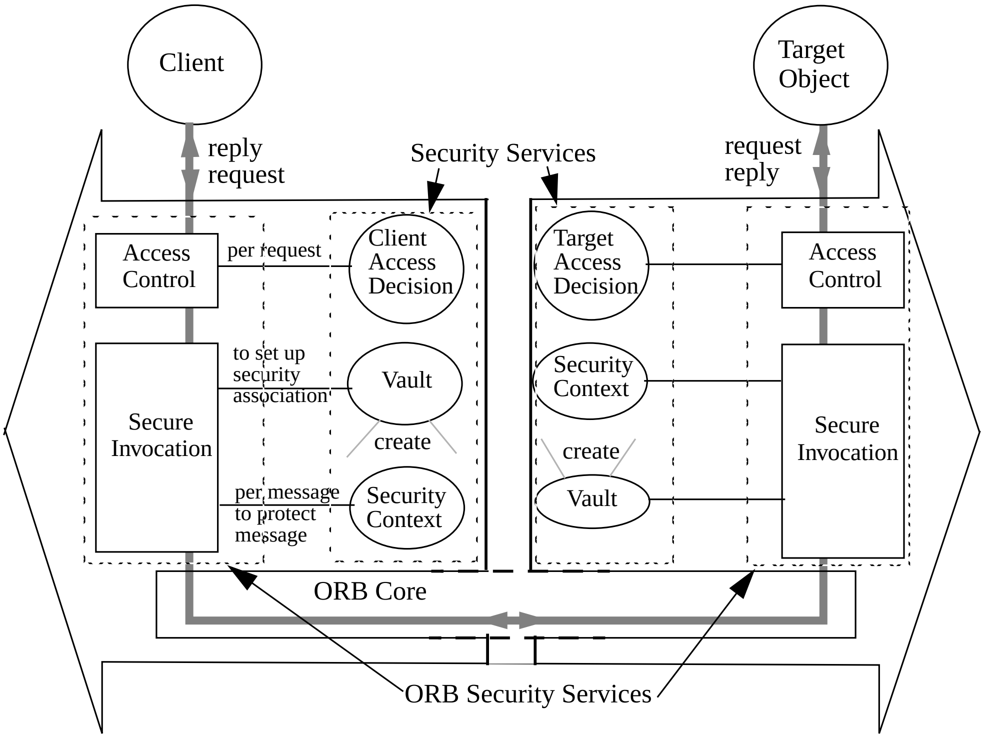


Figure 2-47 ORB Security Services

Two ORB Security Services are shown:

1. The access control service, which is responsible for checking if this operation is permitted and enforcing the invocation audit policy for some event types.
2. The secure invocation service. On the client’s initial use of this object, it may need to establish a security association between client and target object. It also protects the application requests and replies between client and target object.

The security services they use are discussed next.

Access Policy

An **Access Decision** object is used to determine if a given operation on a specific target object is permitted. It is obtained by the ORB service using the **access_decision** attribute of the **Current** object. Since the **Access Decision** objects are **locality constrained**, of necessity the access decision objects at the client and target are distinct.

The ORB service invokes the **access_allowed** operation on the **Access Decision** object specifying the operation required, the principal credentials to be used for deciding if this access is allowed, etc. This is independent of the type of access control policy, which may be discretionary using ACLs or capabilities, mandatory labels usage, etc.

The **Access Decision** object uses the access policy to decide what rights the principal has by invoking the **get_effective_rights** operations on the appropriate **Access Policy** object.

If the access policies use **rights** (rather than directly identifying that this operation is permitted), the **Access Decision** object now invokes **get_required_rights** on the **RequiredRights** object to find what rights are needed for this operation. It compares these rights with the effective rights granted by the policy objects, and if required rights have been granted, it grants access. This model could be extended in the future to handle overlapping access policy domains as described in Appendix F, “Facilities Not in This Specification.”

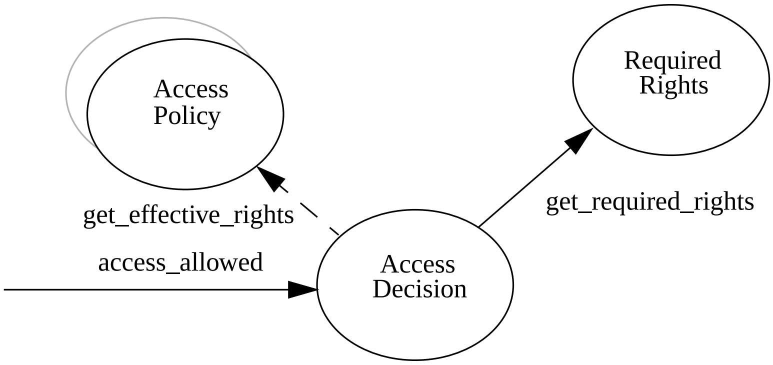


Figure 2-48 Access Decision Object

Vault

The **Vault** object is responsible for establishing the security association between client and target. It is invoked by the Secure Invocation ORB Service at the client and at the target (using **init_security_context** and **accept_security_context)**. The **Vault** creates the security context objects, which are used for any further security operations for this association.

Authentication of users (and some other principals) is done explicitly using the **authenticate** operation described in Section 2.3.3, “Authentication of Principals,” on page 2-73. Authentication of an intermediate object in a chain (or the principal representing the object) may be done automatically by the **Vault** when an intermediate object invokes another object.

The **Vault**, like the security context objects it creates, is invisible to all applications.

Security Context

For each security association, a pair of **Security Context** objects (one associated with the client, and one with the target) provide the security context information. Establishing the security contexts may require several exchanges of messages containing security information, for example, to handle mutual authentication or negotiation of security mechanisms.

Security Context objects maintain the state of the association, such as the credentials used, the target's security name, and the session key. The **is_valid** and **refresh** operations are supported to check the validity of the context and refresh it if possible.

Security Context objects provide operations for protecting messages for integrity and confidentiality such as **protect_message, reclaim_message**.

They also have the **received_credentials** attribute, which is made available via the **Current** object.

A security context can persist for many interactions and may be shared when a client invokes several target objects in the same trusted identity domain. Although neither the client nor target is aware of an "association," it is an important optimizing concept for the efficient provision of security services.

Relationship between Implementation Objects for Associations

There is not always a one-for-one relationship between client-target object pairs and security contexts. For example, if a client uses different privileges for different invocations on that object, this will result in separate security contexts. Also, a security context may be shared between this client's calls on more than one target object. This is normally the case if the target objects share a security name, as shown in Figure 2-49 on page 2-68. Note that the **Vault** decides whether to use the same or a different security context based on the target security name (which may be the name of an object or trusted identity domain).

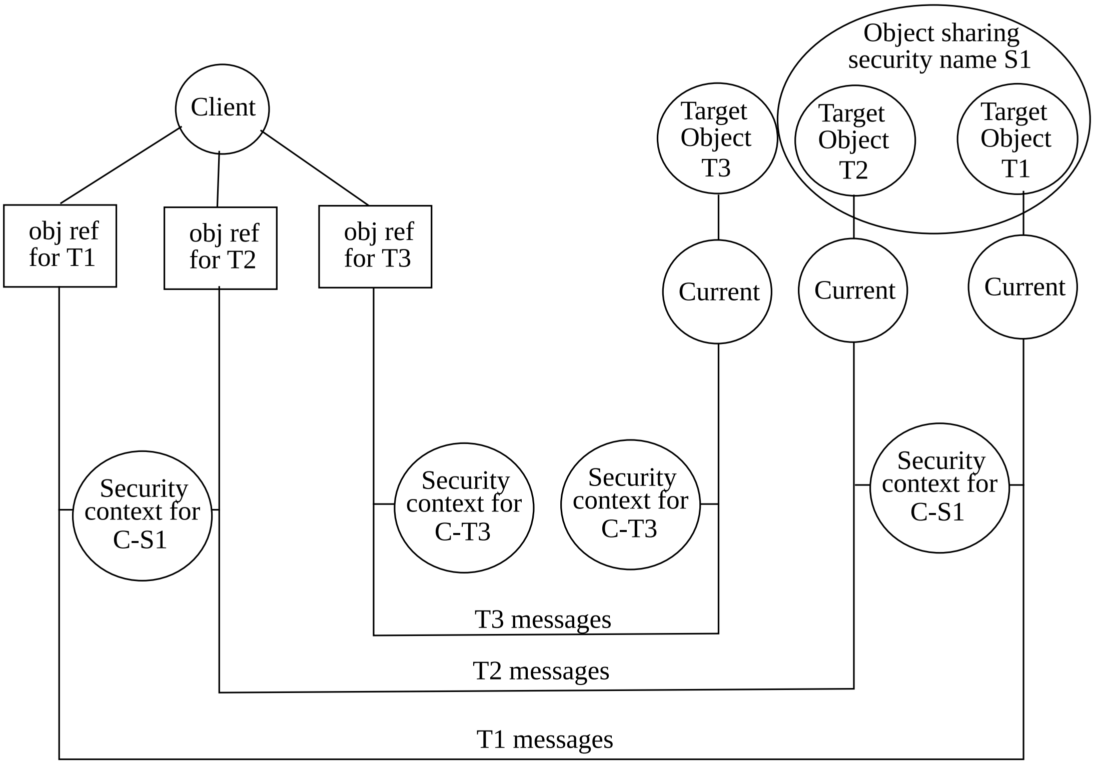


Figure 2-49 Target Objects Sharing Security Names

Implementor's View of Secure Object Creation

When an object is created in a secure environment, it is associated with Security Policy, Environment, and Technology domains as described in Section 2.2.5.2, "Administrative Model," on page 2-58.

The way it is associated with Environment and Technology domains is ORB implementation-specific, and therefore not described here.

For policy domains, the construction policy of the application or factory creating the object is used as shown in Figure 2-50 on page 2-69.

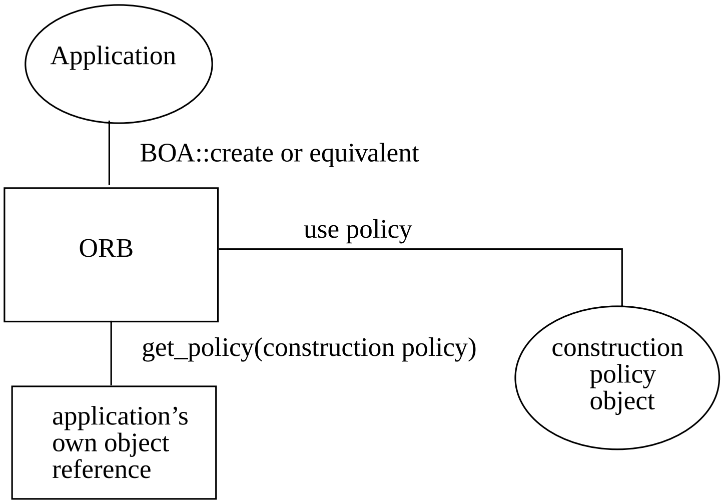

Application
BOA::create or equivalent
ORB
use policy
get_policy(construction policy)
application's own object reference
construction policy object

Figure 2-50 Object Created by Application or Factory

The application (which may be a generic factory) object calls **POA::create_reference** or equivalent to create the new object reference. The ORB obtains the construction policy associated with the object reference to be created. If the application that is attempting to create the object reference is itself a CORBA object, then the ORB attempts to obtain the construction policy associated with it. If the ORB is unable to obtain a construction policy for the object reference to be created, it uses a default construction policy, which does not create a new domain.

The construction policy controls whether, in addition to creating the specified new object reference, the ORB must also create a new domain. If a new domain is needed, the ORB creates both the requested object reference and a domain manager object.

If a new domain is not needed and the application is itself not an object and hence has no domain associated with it, the ORB uses a default domain to place the newly created object reference. In all cases a reference to the domain manager associated with the newly created object reference can be obtained by calling **get_domain_managers** on the newly created object's reference (See the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*).

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain, or an ORB specific default set of policies in the case that the object reference was created in a situation where there is no enclosing domain (e.g., by an application that is itself not a CORBA object and hence has no domain associated with it).

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management operations. Please note: these operations do not form a part of this specification.

2.2.5.4 *Summary of Objects in the Model*

The previous sections have described the various security-related objects, which are available to applications, administrators, and implementors.

Figure 2-51 shows the relationship between the main objects visible in different views for three types of security functionality.

1. Authentication of principals and security associations (which includes authentication between clients and targets) and message protection.
2. Authorization and access control (i.e., the principal being authorized to have privileges or capabilities and control of access to objects).
3. Accountability -- auditing of security-related events and using non-repudiation to generate and check evidence of actions.

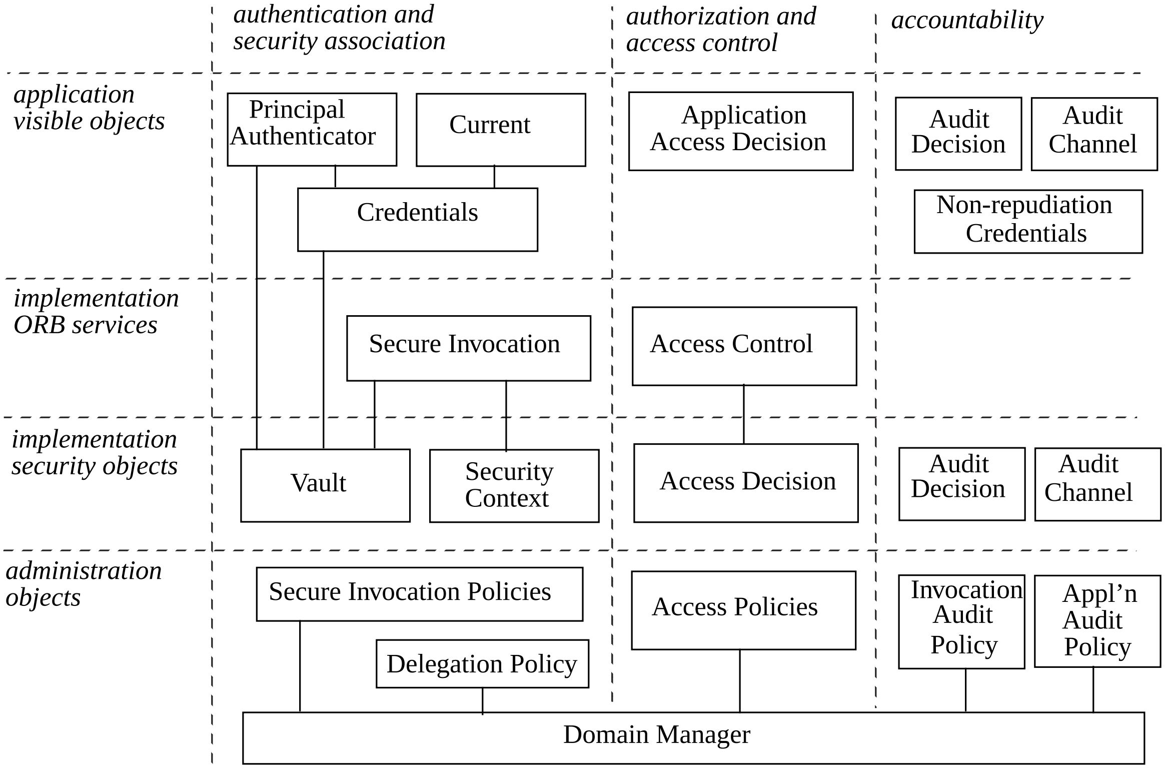


Figure 2-51 Relationship Between Main Objects

Credentials are visible to the application after authentication, for setting or obtaining privileges and capabilities, for access control, and are available to ORB service implementors. Only the first of these usages is shown.

Policy objects have management operations to allow policies to be maintained. These operations depend on the type of policy. For example, management of a mandatory access control policy using labels is different from management of an ACL. However, at run-time, an access decision object is used, which has a standard "check if access is allowed" operation, whatever the access control policy used. The access policy object has the management operations, whereas the access decision object has the runtime decision operations.

The diagram does not show:

- Application objects (client, target object, target object reference at the client).
- The ORB core (though the security ORB services it calls are shown).
- The construction policy object.

2.3 Application Developer's Interfaces

2.3.1 Introduction

This section defines the security interfaces used by the application developer who implements the business logic of the application. For an overview of how these interfaces are used, see Section 2.2.1.3, "Application Developer View," on page 2-30.

Please note that applications may be completely unaware of security, and therefore not need to use any of these interfaces. In general, applications may have different levels of security awareness. For example:

- Applications unaware of security, so that an application object, which has not been designed with security in mind, can participate in a secure object system and be subject to its controls such as:
- Protection default quality of protection on object invocations.
  - Control of who can perform which operations on which objects.
  - Auditing of object invocations.
- Applications performing security-relevant activities. An application may control access and audit its functions and data at a finer granularity than at object invocation.
- Applications wanting some control of the security of its requests on other objects, for example, the level of integrity protection of the request in transit.
- Applications that are more sophisticated in how they want to control their distributed operations, for example, control whether their credentials can be delegated.
- Applications using more specialist security facilities such as non-repudiation.

Security operations use the standard CORBA exceptions. For example, any invocation that fails because the security infrastructure does not permit it, will raise the standard CORBA::NO_PERMISSION exception. A security operation that fails because the feature requested is not supported in this implementation will raise a CORBA::NO_IMPLEMENT exception. Any parameter that has inappropriate values should be flagged by raising the CORBA::BAD_PARAM exception. No security-specific exceptions are specified.

2.3.1.1 Security Functionality Packages

Two security functionality packages and one optional security functionality package are defined in this specification. In addition, the Security Ready functionality packages are also described in this and the two following sections.

Security Functionality Level 1 Package

Security functionality Package 1 provides an entry level of security functionality that applies to all applications running under a secure ORB, whether aware of security or not. This includes security of invocations between client and target object, message protection, some delegation, access control, and audit.

The security functionality is in general specified by administering the security policies for the objects, and is mainly transparent to applications.

Security Functionality Level 1 Package includes operations for applications as follows: **Current::get_attributes** allows an application to obtain the privileges and other attributes of the principal on whose behalf it is operating. It can then use these to control access to its own functions and data (see Section 2.3.4, "The Credentials Object," on page 2-78, and Section 2.3.10, "Access Control," on page 2-103).

Security Functionality Level 2 Package

This security functionality level provides further security functionality such as more delegation options.

It also allows an application aware of security to have more control of the enforcement of this security. Most of the interfaces specified in this section are only available as part of this functionality level. Note that although implementations must support all Level 2 interfaces in order to conform to Security Functionality Level 2, different implementations of these interfaces may support different semantic extensions, while maintaining the same core semantics; some implementations will therefore be capable of enforcing a wider variety of policies than others.

Optional Functionality Package

The only specified optional facility specified here is non-repudiation. The interfaces for this are specified in Section 2.3.12, "Non-repudiation," on page 2-108.

It is possible to add other security policies to this specification, for example, extra access or delegation policies, but these are not part of this specification.

2.3.1.2 Introduction to the Interfaces

The interfaces specified here, as in other sections, are designed to allow a choice of security policies and mechanisms. Where possible, they are based on international standard interfaces. Several of the operations in the **Credentials** interface are based on those of GSS-API.

Data Types

Many of the security data types used by applications are also used for implementation interfaces; therefore, these are defined in a separate module called **Security.** See Section B.2, "General Security Data Module," on page B-1 for the details of the data types used by the interfaces.

Some data types, such as security attributes and audit events, have an extensible set of values, so the user can add values as required to meet user-specific security policies. In these cases, a family is identified, and then a set of types or values for this family. Family identifiers 0-7 are reserved for OMG-defined families, and therefore standard values. More details of these families and associated data types are given in Appendix B, Section B.11, "Values for Standard Data Types," on page B-26.

In the interface specifications in the rest of this section, data types defined in module **Security** are included without the qualifying **Security::** for ease of readability. The full definitions are included in Appendix B.

2.3.2 Finding Security Features

2.3.2.1 Description of Facilities

An application can find out what security facilities this implementation supports, for example, which security functionality level and options it supports. It can also find out what security technology is used to provide this implementation.

The **CORBA::ORB::get_service_information** operation is used to determine what security features are supported by this ORB (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*). To request information about Security service the **CORBA::ServiceType** constant value, **CORBA::Security** should be used. To see what the definition of various service options relevant to security are see the constant definitions of type **CORBA::SecurityOptions** in the IDL Security module in Appendix B, Section B.2, "General Security Data Module," on page B-1.

2.3.3 Authentication of Principals

2.3.3.1 Description of Facilities

A principal must establish its credentials before it can invoke an object securely. For many clients, there are default credentials, created when the user logs on. This may be performed prior to using any object system client. These default credentials are automatically used on object invocation without the client having to take specific action. Even if user authentication is executed within the object system, it should normally be done by a user sponsor/login client, which is separate from the business application client, so that business applications can remain unaware of security.

In most cases, principals must be authenticated to establish their credentials. However, some services accept requests from unauthenticated users. In this case, if the principal has no credentials at the time the request is made, unauthenticated credentials are created automatically for it.

If the user (or other principal) requires authentication and has not been authenticated prior to calling the object system, the (login) client must invoke the **Principal Authenticator** object to authenticate, and optionally select attributes for, the principal for this session. This creates the required **Credentials** object and makes it available as the default credentials for this client. Its object reference is also returned so it can be used for other operations on the **Credentials**. If the object system supports non-repudiation, the credentials returned can be used for non-repudiation operations as specified in Section 2.3.12, "Non-repudiation," on page 2-108.

Authentication of principals may require more than one step, for example, when a challenge/response or other multistep authentication method is used. In this case, the authentication service will return information to the caller, which may be used in further interactions with the user before continuing the authentication. So there are both **authenticate** and **continue_authentication** operations of the **Principal Authenticator** object.

There is no need for an application to explicitly authenticate itself to act as an initiating principal prior to invoking other objects, as this will be performed automatically if needed. However, it does need to be performed explicitly if the object wants to specify particular attributes.

The Principal Authenticator object creates a **Credentials** object and places it on the **Current** object's **own_credentials** list only after **authenticate** or **continue_authentication** returns a value of '**SecAuthSuccess**.**'** The Principal Authenticator always places new credentials at the beginning of the **own_credentials** list. The application may remove **Credentials** objects from the **own_credentials** list with the **Current::remove_own_credentials** operation.

The Principal Authenticator object is a **locality constrained** object.

2.3.3.2 The SecurityLevel2::PrincipalAuthenticator Interface

This section describes the **PrincipalAuthenticator** interface that has following operations.

get_supported_authen_methods

This operation returns the authentication methods that are valid for a particular mechanism that the Vault object supports. This operation raises a CORBA::BAD_PARAM exception if the vault does not support the mechanism.

```
AuthenticationMethodList get_supported_authen_methods(
    in      MechanismType           mechanism
);
```

Parameters

mechanism	Contains the mechanism for which the authentication methods are valid.

Return Value

The list of authentication methods supported by this PrincipalAuthenticator object for the particular mechanism.

authenticate

This operation is called to authenticate the principal and optionally request privilege attributes that the principal requires during its capsule specific session with the system. It creates a capsule specific **Credentials** object including the required attributes and is placed on the **Current** object's **own_credentials** list according to the credential's mechanism type.

```
AuthenticationStatus authenticate(
    in      AuthenticationMethod    method,
    in      MechanismType           mechanism;
    in      SecurityName            security_name,
    in      Opaque                  auth_data,
    in      AttributeList           privileges,
    out     Credentials             creds,
    out     Opaque                  continuation_data,
    out     Opaque                  auth_specific_data
);
```

Parameters

method	The identifier of the authentication method used
mechanism	The security mechanism with which to create the **Credentials**.
security_name	The principal's identification information (e.g., login name).
auth_data	The principal's authentication information such as password or long term key.
privileges	The privilege attributes requested.

creds This parameter contains the **locality constrained** object reference of the newly created **Credentials** object. It is usable and placed on the **Current** object's **own_credentials** list only if the return value is '**SecAuthSuccess**.'

auth_specific_data Information specific to the particular authentication service used

continuation_data If the return parameter from the authenticate operation is '**SecAuthContinue**,' then this parameter contains challenge information for authentication continuation.

Return Value

The return parameter is used to specify the result of the operation.

'SecAuthSuccess' Indicates that the object reference of the newly created initialized credentials object is available in the **creds** parameter.

'SecAuthFailure' Indicates that authentication was in some way inconsistent or erroneous, and therefore credentials have not been created.

'SecAuthContinue' Indicates that the authentication procedure uses a challenge/response mechanism. The **creds** contains the object reference of a partially initialized **Credentials** object. The **continuation_data** indicates details of the challenge.

'SecAuthExpired' Indicates that the authentication data contained some information, the validity of which had expired (e.g., expired password). **Credentials** have therefore not been created.

continue_authentication

This operation continues the authentication process for authentication procedures that cannot complete in a single operation. An example of this continuation is a challenge/response type of authentication procedure.

```
AuthenticationStatus continue_authentication(
    in      Opaque              response_data,
    in      Credentials         creds,
    out     Opaque              continuation_data,
    out     Opaque              auth_specific_data
);
```

Parameters

response_data	The response data to the challenge.
creds	Reference of the partially initialized Credentials object. The Credentials object is fully initialized only when return parameter is '**SecAuthSuccess**.'
continuation_data	If the return parameter from the continue_authentication operation is '**SecAuthContinue**,' then this parameter contains challenge information for authentication continuation.
auth_specific_data	Information specific to the particular authentication service used.

Return Value

The return parameter is used to specify the result of the operation.

'SecAuthSuccess'	Indicates that the **Credentials** object whose reference was identified by the **creds** parameter is now fully initialized.
'SecAuthFailure'	Indicates that the response data was in some way inconsistent or erroneous, and that therefore credentials have not been created.
'SecAuthContinue'	Indicates that the authentication procedure requires a further challenge/response. The **Credentials** object whose reference was identified in the **creds** parameter is still only partially initialized. The **continuation_data** indicates details of the next challenge.
'SecAuthExpired'	Indicates that the authentication data contained some information whose validity had expired (e.g., expired password). The **Credentials** object referred to by the **creds** parameter is not valid.

2.3.3.3 *Portability Implications*

The **authenticate** and **continue_authentication** operations allow different authentication methods to be used. However, methods available are dependent on availability of underlying authentication mechanisms. This specification does not dictate that particular mechanisms should be used. However, use of some mechanisms, (e.g., those involving hardware such as smart cards or finger print readers) may also require use of device-specific objects so the client using such objects will not be portable to systems which do not support such devices. It is therefore recommended that use of both the authenticate operations described here and any device-specific ones be confined to a user sponsor or login client, or that such authentication is done prior to calling the object system, where the credentials resulting from this can be used in portable applications.

2.3.4 The Credentials Object

2.3.4.1 Description of Facilities

A **Credentials** object represents a particular principal's credential information specific to the capsule. It includes information such as that principal's privilege and identity attributes, such as an audit id. (It also includes some security-sensitive data required when this principal is involved in peer entity authentication. However, such data is not visible to applications.)

The **Credentials** object is a **locality constrained** object.

An application may want to:

- Specify security invocation options to be used by default whenever these credentials are used for object invocations.
- Modify the privilege and other attributes in the credentials, for example, specify a new role or a capability. This can modify the current privileges in use, or the application can make a copy of the **Credentials** object first, and then modify the new copy.
- Inquire about the security attributes currently in the credentials, particularly the privilege attributes.
- Check if the credentials are still valid or if they have timed out, and if so, refresh them.

Credentials objects are created as the result of:

- Authentication (see "Authentication of Principals" on page 2-73).
- Copying an existing **Credentials** object.
- Asking for a **Credentials** object via **Current** (see Section 2.3.7, "Security Operations on Current," on page 2-93).

The way these credentials are made available for use in invocations is described in Section 2.2, "Security Architecture," on page 2-28, and defined in detail in Section 2.3.6, "Operations on Object Reference," on page 2-85, and Section 2.3.7, "Security Operations on Current," on page 2-93.

Credentials used for non-repudiation also support further facilities as described in Section 2.3.12, "Non-repudiation," on page 2-108.

2.3.4.2 The SecurityLevel2::Credentials Interface

The following operations are in the **Credentials** interface.

copy

This operation creates a new **Credentials** object, which is an exact duplicate (a "deep copy") of the **Credentials** object which is the target of the invocation. The return value is a reference to the newly created copy of the original **Credentials** object.

Credentials copy();

Parameters

None

Return Value

An object reference to a copy of the Credentials object, which was the target of the call.

destroy

This operation destroys the **Credentials** object that it is invoked on. In general, the caller is always responsible for destroying its copy of the **Credentials** object after it is done with it. When **Credentials** are used as **"in"** parameters the callee always makes a copy if needed. Then onwards the callee is responsible for managing the life-style of the copy that it makes. In case of **Credentials** objects that are returned as result, the caller is responsible for destroying it. In case of **"out"** parameters, the callee is responsible for creating it and the caller is responsible for destroying it. The caller is responsible for providing thread safety for **Credentials** parameters that are passed as **"in"** parameters. They must ensure that no other thread modifies the object until the invoked operation is completed.

void destroy();

Parameters

None

Results

None. The Credentials object is destroyed.

set_privileges

This is used to request a set of privilege attributes (such as role, groups), updating the state of the supplied **Credentials** object. One of the attributes requested may be an attribute set reference, which causes a set of attributes to be requested.

Note – This operation can only be used to set privilege attributes. Other attributes, such as the audit identity, are generated by the system and cannot be changed by the application.

```
boolean set_privileges(
    in      boolean             force_commit,
    in      AttributeList       requested_privileges,
    out     AttributeList       actual_privileges
);
```

Parameters

force_commit	If true, the attributes should be applied immediately; otherwise, attribute acquisition may be deferred to when required by the system
requested_privileges	A set of (typed) privilege attribute values. One of these may be a role name, which is an attribute set reference used to select a set of attributes. (A null attribute set requests default attributes.) Attributes can include capabilities
actual_privileges	The set of (typed) privileges actually obtained

Return Value

TRUE	Indicates that attributes can be set, and that the **actual_privileges** parameter contains the complete set or subset of those attributes requested. It is the responsibility of the application programmer to interrogate the returned attributes to determine their suitability.
FALSE	Operation failed, **Credentials** were not modified

get_attributes

This is used to get privilege and other attributes from the **Credentials**. It can be used to:

- Get privilege attributes, including capabilities, for use in access control decisions. If the principal was not authenticated, only one privilege attribute is returned. This has type *Public* and no meaningful value.
- Get other attributes such as audit or charging identities if available. (If the principal is not authenticated, none of these are returned.)

```
AttributeList get_attributes(
    in      AttributeTypeList          attributes
);
```

Parameters

attributes	The set of security attributes (privilege attributes and identities) whose values are desired. If this list is empty, all attributes are returned.

Return Value

The requested set of attributes reflecting the state of the **Credentials**.

is_valid

Credentials objects may have limited lifetimes. This operation is used to check if the **Credentials** are still valid.

```
boolean is_valid(
    out         UtcT    expiry_time
);
```

Parameters

expiry_time	The time that the **Credentials** expire.

Return Value

TRUE	The **Credentials** is still valid
FALSE	The **Credentials** is not valid anymore

refresh

This operation allows the application to update **Credentials**. Depending on the mechanism, some **Credentials** may need to be refreshed before they expire; may be able to be refreshed after they expire; or may not be able to be refreshed. If **Credentials** cannot be refreshed due to the limitations of the implementation a CORBA::NO_IMPLEMENT exception is raised. If the **Credentials** object cannot be refreshed due to the limitations of the security mechanism a CORBA::BAD_OPERATION exception is raised. If the **Credentials** object cannot be refreshed due to invalid **refresh_data** (i.e. stipulating a new expiry time beyond a legal limit) a CORBA::BAD_PARAM exception is raised.

```
boolean refresh(
    in  Opaque  refresh_data
);
```

Parameters

refresh_data	Data needed to refresh **Credentials**, which is specific to the mechanism type.

get_security_feature

This operation returns a boolean value that represents the value of the given security feature for the given communication direction that the **Credentials** object is supporting.

The communication direction parameter indicates which set of security features (i.e. those set for the request direction, the reply direction, or both) should be returned. Conforming implementations are not required to support the "**request**" and "**reply**" directions. If an unsupported direction is passed to **get_security_feature**, the CORBA::BAD_PARAM exception is raised.

The **get_security_feature** operation has the following definition:

```
boolean get_security_feature(
   in     CommuncationDirection   direction,
   in     SecurityFeature          feature
);
```

Parameters

direction	The communication direction (i.e. both, request, or reply) to which the security feature is applicable. Normally set to both.
feature	The feature for which the value is sought.

Return Value

The boolean value of the security feature supported by the **Credentials** object.

credentials_type

This readonly attribute specifies whether the **Credentials** object is of the “own” credentials type (i.e., created by the **PrincipalAuthenticator**) or it is of the “received” credentials type (i.e., established as the result of a thread specific secure association with a client in the context of servicing a request). It has the following definition:

```
readonly attribute Security::InvocationCredentialsType credentials_type;
```

authentication_state

This readonly attribute specifies the authentication state the **Credentials** object. For **Credentials** that are created by the **PrincipalAuthenticator**, this attribute tells whether the **Credentials** are partially initialized. It has the following definition:

```
readonly attribute Security::AuthenticationStatus authentication_state;
```

Values

’SecAuthSuccess’	**Credentials** are fully initialized. **Credentials** may be valid.
’SecAuthFailure’	Authentication has failed. **Credentials** are invalid. **Credentials** may be in this state if they were partially initialized in a call to **PrincipalAuthenticator::authenticate** and then failed in the **PrincipalAuthenticator::continue_authentication** operation.
’SecAuthContinue’	**Credentials** are partially initialized. **Credentials** that are not yet valid for use.
’SecAuthExpired’	**Credentials** initialization has expired. Credentials are invalid.

mechanism

This readonly attribute specifies the mechanism the **Credentials** object represents. It has the following definition:

```
readonly attribute MechanismType mechanism;
```

accepting_options_supported and accepting_options_required

These two attributes are the options that the **Credentials** object support and require to accept secure associations from clients. These two attributes can be thought of as directly relating to the **target_supports** and **target_requires** association options attributes that may be advertised in a security mechanism component in a target object's IOR. Section 3.1.4.1, "Security Components of the IOR," on page 3-8

Note – Not all mechanisms may use such a security component in IOR.

When the **Credentials** are created by the **PrincipalAuthenticator** these options will be set to default values depending on initialization scheme of the particular mechanism. Authentication data may contain constraints on the supported/required association options as well as constraints on the mechanism itself.

Setting these attributes to values that are invalid for the mechanism raises a CORBA::BAD_PARAM exception. In general, the **accepting_options_required** cannot be set to have "more" capability than the **accepting_options_supported** and the **accepting_options_supported** cannot be set to have "less" capability than the **accepting_options_required**.

These attributes have the following definition:

```
attribute AssociationOptions        accepting_options_supported;
attribute AssociationOptions        accepting_options_required;
```

invocation_options_supported and invocation_options_required

This attribute is used to control the security characteristics of the secure association by which these **Credentials** are used to make an invocation on a target object. These association options affect the characteristics of a secure association setup, such as the delegation mode to use, whether trust in the target is needed, and the message protection is required.

Setting this attribute to an invalid value, which may be constrained by the mechanism or the internal state of the **Credentials**, will raise a CORBA::BAD_PARAM exception.

This attribute has the following definition:

```
attribute AssociationOptions invocation_options_supported;
attribute AssociationOptions invocation_options_required;
```

2.3.5 The ReceivedCredentials Object

2.3.5.1 Description of Facilities

A **ReceivedCredentials** object represents a remote principal's credential information for a secure association and therefore includes much of the same information as in an "own" type **Credentials** object, such as the principal's privilege attributes and identities. **ReceivedCredentials** may also be used for invocations (delegation). Therefore, the **ReceivedCredentials** interface inherits from the **Credentials** interface.

A **ReceivedCredentials** object represents the secure association to the application. Therefore, the **ReceivedCredentials** object contains the properties of that association, such as the **Credentials** local to the capsule used for the association, the association options in effect, the delegation state of the remote principal, and the delegation mode of the **ReceivedCredentials**.

A **ReceivedCredentials** object, since it represents a secure association, may have a lifetime associated with a single thread of execution servicing a request. It is retrieved from the security **Current** object through the **received_credentials** attribute.

ReceivedCredentials object is a **locality constrained** object, and it contains a **credentials_type** value of **SecReceivedCredentials**.

2.3.5.2 The SecurityLevel2::ReceivedCredentials Interface

The **ReceivedCredentials** interface is defined as follows:

```
interface ReceivedCredentials : Credentials { // Locality Constrained
    readonly attribute Credentials              accepting_credentials;
    readonly attribute AssociationOptions       association_options_used;
    readonly attribute DelegationState          delegation_state;
    readonly attribute DelegationMode           delegation_mode;
};
```

accepting_credentials

This readonly attribute contains the reference to the credentials that are used on the accepting side of the negotiation of the secure association with the remote principal.

association_options_used

This readonly attribute contains the association options in effect for the secure association with the remote principal.

delegation_state

This readonly attribute tells the delegation state of the remote principal for these credentials. It has the following values:

Values

’SecInitiator’	The remote principal is the acting in his own behalf.
’SecDelegate’	The remote principal is acting in behalf of another principal

Note – Not all security mechanisms may be able to indicate if the remote principal is a delegate. For example, with unrestricted delegation, sometimes known as *impersonation*, the value of this attribute would always be **SecInitiator**.

delegation_mode

This readonly attribute indicates the delegation mode of the credentials. It has the following values.

Values

‘SecDelModeNoDelegation’	The credentials cannot be used to make invocations.
‘SecDelModeSimpleDelegation’	The credentials can be used to make invocations with no traced capability.
‘SecDelModeCompositeDelegation’	The credentials can be used to make invocations with some composite delegation scheme.

2.3.5.3 Portability Implications

The **PrincipalAuthenticator::authenticate** and **Credentials::set_privileges** operations allow particular privilege attributes to be specified. The attributes supported by different systems may vary according to security policies supported. It is recommended that use of these interfaces be limited, so business application objects are not exposed to particular policy details (unless they need to be, as they are enforcing compatible security policies directly).

2.3.6 Operations on Object Reference

2.3.6.1 Description of Facilities

If the client application is unaware of security (for example, was written to use an ORB without security), the ORB services will enforce the relevant security policies transparently to applications. As described elsewhere, the security enforced is specified by:

- The security policy set at the client by administrative action.
- The credentials used by the client.
- The security policy for the target object. Relevant security information about this is made available to the client in the target's object reference.

These policies include association options, any controls on whether this client can perform this operation on this target, and the quality of protection of messages.

The only visibility of security to most applications is that some operations will now fail because they would breach security controls.

An application client unaware of security can communicate with a security aware one and vice versa.

A client application aware of security can also specify what security policy options it wants to apply when communicating with this target object by performing operations on the target object's reference and the binding object associated with it. The following operations are available on the target object reference.

- **get_policy** is used to find the policy of the specified type (including those relevant to security) for this object.
- **get_domain_managers** is used to obtain a list of domain managers that the given object is associated with.
- **set_policy_overrides** is used to set overrides of default policies on individual object references.

Although these operations are on the target object reference, the scope of the effect of the operation is the use of that reference itself, and not the object that it represents. That is, the act of obtaining a copy of an object reference with a new set of override policies set on it in no way affects the target object that the object reference in question is associated with.

A target object can influence the security policy for incoming invocations by setting security policies using the administrative operations in Section 2.4, "Administrator's Interfaces," on page 2-116. This will affect the security information exported as part of its object reference.

The default policies that can be overridden using the **set_policy_overrides** operation are:

- **QOP** - the quality of protection that will be provided to any successful invocation using that object reference. The **QOPPolicy** object is the bearer of this policy.
- **Invocation Credentials** - the Credentials that will be used in invocations using that object reference. The **InvocationCredentialsPolicy** object is the bearer of this policy.
- **Security Mechanisms** - the mechanisms (one of) which must be used for successful invocation using the object reference. The **MechanismsPolicy** object is the bearer of this policy.

- **Establish Trust** - the directive for the establishment of trust of client by target and target by client. The **EstablishTrustPolicy** object is the bearer of this policy.
- **Delegation Directive** - the directive telling whether delegation should be used during the invocation. The **DelegationDirectivePolicy** object is the bearer of this policy.

The above policy objects can be created using the **ORB::create_policy** operation. The above policy objects must be put in a **PolicyList** and given to the **set_policy_overrides** operation on the target object reference. If successful, the operation returns a new object reference that uses the new policy overrides for subsequent invocations.

The policies currently associated with the object reference, including overridden ones can be accessed using the **get_policy** operation. This operation returns a Policy object of the appropriate type containing the current policy, which can be extracted from the readonly attribute in the Policy object interface.

Note – The application states its **minimum** security requirements. A higher level of security may still be enforced as this may be required by security policy. Thus operationally the default policies will actually be overridden only if the requested overrides are consistent with the overall security policy.

2.3.6.2 *Client Side Invocation Policy Objects*

There are a number of Policy objects that are bearers of the client side invocation related policies. They are as follows:

QOP Policy

The QOP Policy object has a policy type of **Security::SecQOPPolicy** and has the **QOPPolicy** interface, which is shown below.

```
interface QOPPolicy : CORBA::Policy {          // Locality Constrained
    readonly attribute Security::QOP        qop;
};
```

This interface has a single readonly attribute `qop` which represents the policy in the form of an enum value of type **Security::QOP**.

This object can be passed to **set_policy_overrides** to specify that a particular quality of protection is required for messages sent using the object reference returned by the **set_policy_overrides** operation. When this object is returned by the **get_policy** operation it contains the quality of protection policy associated with this object reference.

Mechanism Policy

The Mechanism Policy object has a policy type of **Security::SecMechanismPolicy** and has the **MechanismPolicy** interface, which is shown below.

```
interface MechanismPolicy : CORBA::Policy {// Locality Constrained
      readonly attribute Security::MechanismTypeList  mechanisms;
};
```

This interface has a single readonly attribute **mechanisms**, which represents the policy in the form of a **Security::MechanismTypeList**.

This object can be passed to **set_policy_overrides** to request the use of one of a specific set of mechanisms in invocation through the object reference returned by the **set_policy_overrides** operation. When this object is returned by **get_policy** it contains the security association mechanisms available through this object reference.

Invocation Credentials Policy

The Invocation Credentials Policy object has a policy type of **Security::SecInvocationCredentialsPolicy** and has the **InvocationCredentialsPolicy** interface, which is shown below.

```
interface InvocationCredentialsPolicy : CORBA::Policy { // Locality Constrained
      readonly attribute CredentialsList            creds;
};
```

This interface has a single readonly attribute **creds** which returns a list of **Credentials** objects which will be used as invocation credentials for invocations through this object reference.

This object can be passed to **set_policy_overrides** to specify one or more **Credentials** objects to be used when calling this target object using the object reference returned by **set_policy_overrides**. For example, the client may want to make different privileges available to different targets by choosing **Credentials** with the required privileges. When this object is returned by **get_policy** it contains the active credentials that will be used for invocations via this target object reference.

Establish Trust Policy

The Establish Trust Policy object has a policy type of **Security::EstablishTrustPolicy** and has the **EstablishTrustPolicy** interface, which is shown below.

```
interface EstablishTrustPolicy : CORBA::Policy { // Locality Constrained
       readonly attribute EstablishTrust trust;
};
```

This interface has two readonly attributes.

trust

This attribute is a structure that contains two attributes each stipulating whether trust in client and trust in target is enabled.

- The **trust_in_client** element of this attribute stipulates whether the invocation must select credentials and mechanism that will allow the client to be authenticated to the target. (Some mechanisms may not support client authentication).

- The **trust_in_target** element of this attribute stipulates whether the invocation must first establish trust in the target.

This object can be passed to **set_policy_overrides** to specify that a particular trust policy be followed for invocations using this object reference. When this object is returned by the **get_policy** operation it contains the trust policy associated with this object reference.

Delegation Directive Policy

The Delegation Directive Policy object has a policy type of **Security::DelegationDirective** and has the **DelegationDirectivePolicy** interface, which is shown below.

```
interface DelegationDirectivePolicy : CORBA::Policy { // Locality Constrained
    readonly attribute Security::DelegationDirective    delegation_directive;
};
```

This interface has a single readonly attribute **delegation_directive** that represents the policy stating whether delegation should be used when making invocations on an object. If the policy states that delegation should be used, then the **Credentials** object selected for the invocation must support delegation.

This object can be passed to **set_policy_overrides** to specify that a delegation policy be followed for invocations using this object reference. When this object is returned by the **get_policy** operation it contains the delegation policy associated with this object reference.

2.3.6.3 Semantics of Combined Client Policies

The client side policies that are defined for a particular object reference employ a particular semantics in determining the security characteristics of invocations made with that object reference. When applied to an object reference, the ORB performs a decision procedure to determine the security characteristics that are compatible between the security mechanisms that the target object supports and the client side security policies that are attached to the target object's reference. It is entirely possible that the set of policies when applied to the object reference may be inconsistent. The basic thrust of this decision procedure is to select the proper **Credentials** object from the list of credentials supplied in the **InvocationCredentialsPolicy** object.

The following decision procedure is applied by the security service to eliminate the **Credentials** made available for invocation by list of **Credentials** objects in the **InvocationCredentialsPolicy**. The decision procedure is used amongst this list of **Credentials** objects, the other client side security policies, and the target objects's IOR. This decision procedure determines the security mechanism, a compatible **Credentials** object, and a security component from the target's IOR to use for the invocations made on that object reference. It should be noted that **Credentials** are selected from sequence of **Credentials** returned by the **creds** attribute selector of the **InvocationCredentialsPolicy** object. These credentials are examined first by their mechanism by virtue of the **MechanismPolicy** object, then by the Credentials being able to support other policies that may apply.

It is the goal of the decision procedure to select a single **Credentials** object with which to make the invocation. However, it is entirely possible that constraints provided by other client polices, (such as the **MechanismPolicy**) and the target object's IOR eliminate all **Credentials** objects from the list, thereby raising a CORBA::NO_RESOURCES exception. Also, it is possible that the elimination procedure leaves more than one **Credentials** object. In this case, any of the Credentials objects are viable for making the invocation. However, a selection of a single **Credentials** object still needs to be made. At this point, it is left up to the ORB to select a Credentials object from a list of remaining available credentials.

The elimination decision procedure is as follows:

```
 For each mechanism type in the MechanismPolicy {
   Select a matching security component in the target's IOR by the mechanism
   type.
   If a matching component is found {
     Find a Credentials object in the credentials list that supports the
       mechanism.
     If a Credentials object is found and it supports
       the QOP Policy,
       the Delegation Directive Policy,
       and the Establish Trust Policy {
         If the association options implied by all policies are supported
         by the selected security component in the IOR and all the
         required association options of security component are satisfied {
           Use the selected Credentials and selected attributes to set up the secure
           association.
         } else {
           Find the next credentials object that supports the selected mechanism and
           continue.
         }
       } else {
         Find the next credentials object that supports the selected mechanism and
         continue.
       }
   } else {
     Get the next mechanism type from the MechanismPolicy and continue.
   }
 }
 If no mechanism can be found {
   A CORBA::NO_RESOURCES exception is raised with an informative message.
 }

}
```

2.3.6.4 Security Relevant Operations in the CORBA::Object Interface

These operations are defined in detail in the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*. A brief description is included here to help users of the Security Services.

get_policy

This gets the security policy object of the specified type, which applies to this object. This operation is also available on **Current** and is generally used there to get the policies for the current object.

The **get_policy** operation is used on object references during administration. For example, it may be used to get the policy for a domain.

```
CORBA::Policy get_policy(
    in      CORBA::PolicyType       policy_type
);
```

Parameters

policy_type	The type of policy to be obtained.

Return Value

policy	A policy object of the type specified by the policy_type parameter.

Exceptions

CORBA::BAD_PARAM	Raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

get_domain_managers

get_domain_managers allows administration services (and applications) to retrieve the domain managers, and hence the security and other policies applicable to individual objects that are members of the domain.

```
DomainManagersList get_domain_managers ();
```

Parameters

None.

Return Value

A list of immediately enclosing domain managers of this domain manager. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

set_policy_overrides

set_policy_overrides makes it possible to override a subset of the policies that apply to a specific object reference. It takes two input parameters.

- The first parameter **policies** is a sequence of references to **Policy** objects.
- The second parameter **set_add** of type **CORBA::SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**CORBA::ADD_OVERRIDE**) in the object reference, or they should be added to a clean override free object reference (**CORBA::SET_OVERRIDE**). This operation associates the policies passed in the first parameter with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. The associations last until the object reference is destroyed or the process/capsule/ORB instance in which it exists is destroyed.

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with current is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt will fail and return a CORBA::NO_PERMISSION exception.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

Object set_policy_overrides(
    in      PolicyList              policies,
    in      SetOverrideType         set_add
);
```

Parameters

policies	A sequence of Policy objects that are to be associated with the new copy of the object reference returned by this operation.
set_add	Whether the association is in addition to (**ADD_OVERRIDE**) or as replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

2.3.6.5 Portability Implications

The security features that can be set are generally ones supported by a variety of security mechanisms. Applications using them will therefore be portable between any systems where the security mechanisms support these features. However, some security mechanisms will not support all features, for example, they may not provide replay protection, or may not support confidentiality of application data (owing to regulatory controls). Applications should check the response when attempting to set security features, and if a requested feature is not available, take suitable action.

2.3.7 Security Operations on Current

2.3.7.1 Description

The **Current** object represents service specific state information associated with the current execution context (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*); both clients and targets have **Current** objects representing state associated with the thread of execution and the process/capsule in which the thread is executing (their execution contexts).

The operations of the **Current** object is intended to return information pertaining to the state associated with the current execution context. This includes information specific to both the thread of execution that is used to invoke the operation, as well as the process or capsule to which the thread belongs. State changes affecting state that is associated purely with the thread and not with any broader execution context like capsule (i.e., thread specific) is lost, once the operation within the execution of which this was done completes its execution, thus returning the thread to the ORB. State changes to state associated with a broader execution context like a capsule (i.e., capsule specific) on the other hand persists across multiple invocation of operations in the target object, until it is further modified through operations of the **Current** object or by other means.

The **SecurityLevel1::Current** and the **SecurityLevel2::Current** interfaces described in this section contains operations of both types. In this section, each operation is identified to be either thread specific or process specific to distinguish their behavior.

Note that a reference to the **Current** object representing the active execution context can be retrieved using the **ORB::resolve_initial_references(“SecurityCurrent”)** operation (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*). In a secure ORB, the **Current** object includes operations relevant to Security. The **CORBA::Current** object returned by the **resolve_initial_references** operation can be narrowed to **SecurityLevel1::Current** or **SecurityLevel2::Current** as desired.

The operations on the **Current** object are described in this section and provide access to information about one or more of the following credentials:

- **own credentials**: the list of credentials associated with the active application (capsule). A capsule's own credentials are normally set up as the result of the application being initialized or explicitly by calling on the **PrincipalAuthenticator** object.
- **received credentials**: the credentials received from the client of the invocation as seen at the target object.

The operations provided are the following:

- **get_attributes** (thread specific) obtain privilege and other attributes associated with received credentials (which should be the user's privileges when at the workstation).
- **set_credentials** (thread specific) can specify the type of credentials. This changes the credentials to be used in the future for invocation, as its own credentials, or for non-repudiation. These **Credentials** apply only to those object reference in which the invocations credentials have not been overridden.
- **get_credentials** (thread specific) can obtain the credentials currently associated with the **Current** object for invocation, non-repudiation. These credentials apply to those object references in which the invocation credentials have not been overridden.

The application can also use the following:

- **get_policy** (capsule specific) operation to find what security policies apply to it.
- **own_credentials** (capsule specific) attribute containing the credentials owned by the application.
- **received_credentials** (thread specific) attribute containing the credentials received from the client by the application.
- **required_rights_object** (capsule specific) attribute to discover which operations require which rights.
- **principal_authenticator** (capsule specific) attribute to get a reference to the PrincipalAuthenticator object (which can be used to authenticate principals and thus obtain **Credentials** objects for them).
- **access_decision** (capsule specific) attribute to get a reference to the **Access Decision** object.
- **audit_decision** (capsule specific) attribute to get a reference to the **Audit Decision** object.
- **get_security_mechanisms** returns the security mechanism data for the target.

It should be noted that when an application starts its execution and gains access to an ORB using the **ORB_init** operation, it immediately gets a set of default policies and credentials associated with it. Each thread executing in that capsule inherits these defaults and continues to be guided by them until any of the defaults are replaced by the use of a **set_*** operation in the thread. Subsequently the new credentials and features set using the **set_*** operation remain active until they are modified again by further use of **set_*** operations or the thread terminates or leaves the capsule. The corresponding **get_*** operations return the currently active credentials, policies, feature and attributes associated with the thread.

It should further be noted that if the policies associated with any individual object reference has been overridden using the **Object::set_policy_overrides** operation, then the overridden policies take precedence over the corresponding thread policies, when the said thread is used to carry out an object invocation using the said object reference.

2.3.7.2 The SecurityLevel1::Current Interface

The following operations are available in the **SecurityLevel1::Current** interface.

get_attributes

This is thread specific operation that is used to get privilege (and other) attributes from the client's credentials. It is available in the security functionality Level 1 to allow applications to enforce their own security policies without these applications having to perform operations on credentials.

This operation can be used to get:

- Privilege attributes for use in access control decisions. If the principal was not authenticated, only one privilege attribute is returned. This has type **Public** and no meaningful value.
- Other attributes, such as audit or charging identities, if available.

At the client, this generally gets the user's (or other principal's) privileges. At the target, it gets the received privileges.

```
AttributeList get_attributes(
    in      AttributeTypeList           attributes
);
```

Parameters

attributes	The set of security attributes (privilege attributes and identities) whose values are desired. if this list is empty, all attributes are returned.

Return Value

The set of attributes or identities reflecting the state of the **Credentials**.

2.3.7.3 The SecurityLevel2::Current Interface

The following operations are to be found in the **SecurityLevel2::Current** interface.

set_credentials

This operation pertains to the thread specific state associated with the **Current** object. **Credentials** are associated with **Current** for different types of use. **Credentials** are automatically associated with **Current** by the object system at initialization, authentication, and object invocation. However, the application may want to specify particular credentials to use. The **set_credentials** operation sets the specified credentials as the default one for the following:

- Subsequent *invocation*s made by that client. (**SecInvocationCredentials**)
  This may be done to reduce the privileges available to that client by setting credentials having fewer privileges. Also, an intermediate object can explicitly ask for the received credentials to be delegated by using the **Current::received_credentials** as the specified credentials on **set_credentials**.
- Non-repudiation. (**SecNRCredentials**)
  As for the invocation credentials, non-repudiation credentials may be set transparently to the business application. The credentials used for non-repudiation may be the same as the credentials used for invocations. Note that in the previous sentence the word “credentials” is used in the English sense of the word and it does not refer to a **Credentials** object.

```
void set_credentials(
    in      CredentialType      cred_type,
    in      CredentialsList     creds,
    in      DelegationMode      del
);
```

Parameter

cred_type	The type of credential to be set (i.e., **SecInvocationCredentials** or **SecNRCredentials**).
creds	The object reference of the **CredentialsList**, which is to become the default
del	The delegation mode for the credentials being set takes values of **Delegate** or **NoDelegate**.

Return Value

None

get_credentials

This thread specific operation allows an application access to the credentials associated with its execution environment. As for **set_credentials**, the application can ask for the default credentials for future invocations or the ones used for non-repudiation.

An application will normally get invocation or other credentials when it wants to modify them (for example, reduce the privileges available).

```
CredentialsList get_credentials(
    in        CredentialType         cred_type
);
```

Parameters

cred_type	The type of credential to be obtained.

Return Value

A **CredentialsList**.

received_credentials

At a target object, this thread specific attribute is the credentials received from the client. They are the credentials of the principal identified that made the invocation.

In the case of a pure client, for example, an application that is not servicing an invocation on one of its objects (if any), accessing the **received_credentials** attribute causes a CORBA::BAD_OPERATION exception to be raised.

```
readonly attribute ReceivedCredentials received_credentials;
```

Return Value

The **ReceivedCredentials** object reference received from the requester.

supported_mechanisms

This readonly attribute returns the list of supported mechanisms and options supported by the ORB security service. It has the following definition:

```
readonly attribute MechandOptionsList supported_mechanisms;
```

own_credentials

Any application owns a set of credentials which it obtains through the process of authentication of the principal that initiates the execution of the program, and further from other credentials that such a principal might bestow upon the application. This attribute returns this set of credentials.

```
readonly attribute CredentialsList own_credentials;
```

Return Value

A sequence of **Credentials** object references owned by the application.

get_policy

This capsule specific operation returns the policy object of the specified policy_type for the non CORBA object client from which it is invoked, or for the CORBA object from which it is invoked.

```
Policy get_policy(
    in CORBA::PolicyType      policy_type
);
```

Parameters

policy_type	The type of policy to be obtained.

Return Value

A policy object which can be used to interrogate the policy in force as defined in Section 2.4, "Administrator's Interfaces," on page 2-116. For example, the secure invocation policy would give the secure associations defaults for this object, and the delegation policy would say which credentials were delegated on invocations by this object.

required_rights_object

This capsule specific read only attribute is the **RequiredRights** object available in the environment. This object is rarely used by applications directly. It is generally used by **Access Decision** objects to find the rights required to use a particular interface; however, it could be used directly by the application if it wishes to do all its own access control, and base this on **Rights**.

```
readonly attribute RequiredRights required_rights_object;
```

Return Value

An object references to a **RequiredRights** object. The operations in the interface of this object are defined in Section 2.4.4, "Access Policies," on page 2-119.

principal_authenticator

This capsule specific read only attribute is the **PrincipalAuthenticator** object available in the environment. It can be used by the application to authenticate principals and obtain **Credentials** containing their privilege attributes.

```
readonly attribute PrincipalAuthenticator principal_authenticator;
```

Return Value

An object references to a **PrincipalAuthenticator** object. The operations in the interface of this object are defined in Section 2.1.2, "Principals and Their Security Attributes," on page 2-3.

access_decision

This capsule specific read only attribute is the **AccessDecision** object available in the environment. It can be used by the application to obtain decisions regarding accessibility of specific objects from this environment.

```
readonly attribute AccessDecision access_decision;
```

Return Value

An object references to an **AccessDecision** object. The operations in the interface of this object are defined in Section 2.3.10, “Access Control,” on page 2-103.

audit_decision

This capsule specific read only attribute is the **AuditDecision** object available in the environment. It can be used by the application to obtain information about what needs to be audited for the specified object/interface in this environment.

```
readonly attribute AuditDecision audit_decision;
```

Return Value

An object references to an **AuditDecision** object. The operations in the interface of this object are defined in Section 2.3.8, “Security Audit,” on page 2-100.

get_security_mechanisms

This operation is for use by security sophisticated applications. It is used by clients that wish to determine the security mechanisms, security names, and association options that are associated with the target. It is possible for different security names and association options to be used for the target, depending on the mechanism used for the target.

Note – The security name may be shared by several objects.

```
SecurityMechanismDataList get_security_mechanisms(
    in      Object                    obj_ref
);
```

Parameters

obj_ref	The Object reference of the target object of which the security mechanism data is being sought.

Return Value

A list of **SecurityMechanismData** structures, each containing a security mechanism, security name, and association options that are associated with the target object.

remove_own_credentials

This operation is used by applications that wish to remove credentials that were put on the **own_credentials** list by virtue of the **PrincipalAuthenticator**. This operation does not manipulate or destroy the objects in any way. The given **Credentials** object (as opposed to one produced by a **copy** operation) must reside on the list of the **Current** object's **own_credentials**; otherwise, a CORBA::BAD_PARAM exception is raised.

```
void remove_own_credentials(
    in        Credentials                    creds
),
```

Parameters

creds	The **Credentials** object to be removed from the list.

Return Value

None.

2.3.8 Security Audit

2.3.8.1 Description of Facilities

Auditing of object invocations is done automatically by the ORB according to the audit invocation policies (**Security::SecClientInvocationAudit** and **Security::SecTargetInvocationAudit**) for this application.

Applications can also audit their own security relevant activities, where the auditing performed by the ORB does not audit the required activities and/or data.

In this case, the application is responsible for enforcing the application audit policy. It uses an **audit_needed** operation on the **Audit Decision** object for the policy to decide which activities to audit.

Audit information is passed to an **Audit Channel** object in the form of an audit record. The audit record must contain, or be sufficient to identify:

- The type of event.
- The principal responsible for the action, identified by its credentials.
- Event-specific data associated with the event type. This will vary, depending on the event type.
- The time. This may or may not be secure.

It may also want to record some of the values used for selecting whether to audit the event, for example, its success or failure.

An application audit policy will specify the event families and event types as defined in Section 2.4.5, "Audit Policies," on page 2-131.

2.3.8.2 The SecurityLevel2::AuditDecision Interface

The **Audit Decision** object has the **SecurityLevel2::AuditDecision** interface. Its operations described below, help specify what to audit. It is a **locality constrained** object.

The **Audit Decision** object is a **locality constrained** object.

audit_needed

This operation on the **Audit Decision** object is used to decide whether an audit record should be written to the audit channel. The application specifies the event type to be checked and the values for the selectors, which the audit policy requires to make the decision. This operation identifies the interface associated with the audit event using the **InterfaceName** selector value within **value_list**, if defined. If the **InterfaceName** selector value is the empty string, the most derived interface in the **ObjectRef** selector value is used. **ObjectRef** is also used to find the domain containing the relevant audit policy. If **ObjectRef** is not defined, **audit_needed** will not be able to match any **AuditPolicy** and will return false. To ensure that **audit_needed** can match against any potential **AuditPolicy**, the caller must supply all selector values (**ObjectRef**, **Operation**, **Initiator**, and **SuccessFailure**) in **value_list**.

```
boolean audit_needed(
    in AuditEventType               event_type,
    in SelectorValueList            value_list
);
```

Parameters

event_type	Event type associated with the operation.
value_list	List of zero or more selector id value pairs.

Return Value

TRUE	If an audit record should be created and sent to the audit channel.
FALSE	If an audit record is not needed.

audit_channel

This attribute of the **Audit Decision** object provides the audit channel associated with this audit decision object.

```
readonly attribute AuditChannel audit_channel;
```

Return Value

The **Audit Channel** object associated with the **Audit Decision** object.

A standard audit policy is specified in Section 2.4, “Administrator’s Interfaces,” on page 2-116, but if this is to be replaceable without ORB/interceptor changes, a standard interface needs to be available for the ORB or interceptor to call. Therefore, for replaceability, the selectors used on audit needed during invocation must always be the same (see **value_list** above), though not all of these need to be used in taking the decision to audit, depending on policy. Note that the time is not passed over this interface. If the selectors specified in the audit policy use time to decide on whether to audit the event, the **Audit Decision** object should obtain the current time itself.

2.3.8.3 The SecurityLevel2::AuditChannel Interface

The single operation in the **SecurityLevel2::AuditChannel** interface is used to write the audit records. The **Audit Channel** object is a **locality constrained** object.

audit_write

This operation writes an audit record to the **Audit Channel** object, and hence the audit trail. The audit trail is implementation-specific and outside the scope of this chapter. It is expected to be an event service of some sort, such as a CORBA Event Service.

```
void audit_write(
    in      AuditEventType              event_type,
    in      CredentialsList             creds,
    in      UtcT                        time,
    in      SelectorValueList           descriptors,
    in      Opaque                      event_specific_data
);
```

Parameters

event_type	The type of event being audited.
creds	The credentials of the principal responsible for the event. If no credentials are specified, the **own_credentials** attribute associated with **Current** are used.
time	The time the event occurred.
descriptors	A set of values to be recorded associated with the event in the audit trail. These are often the same values as those used to select whether to audit the event.
event_specific_data	Data specific to a particular type of event, to be recorded in the audit trail.

Return Value

None.

audit_channel_id

This is a readonly attribute that contains the id of this audit channel, which is used to identify it in the corresponding audit policy object. This is necessary because the audit channel object itself has to be a **locality constrained** object by virtue of the fact that the **audit_write** operations passes a list of **Credentials**, a **locality constrained** object, as a parameter, while the audit policy object needs to be not thus constrained.

The audit channel identified by the **audit_channel_id** in the **Audit Policy** object is actually associated with the **Audit Channel** interface by the **Audit Decision** object when its **audit_channel** attribute is accessed.

```
readonly attribute AuditChannelId  audit_channel_id;
```

Return Value

audit_channel_id	The channel id of the audit channel.

2.3.8.4 Portability Implications

An application relying on the system audit policies enforced at invocation time is portable to different environments, although the audit policies themselves may need changing.

Applications with their own application audit policies are portable, providing the audit policy itself is portable and the selectors used are available in these environments. For example, if selectors use privileges, the same ones must be available.

2.3.9 Administering Security Policy

When an object is created, it automatically becomes a member of one or more domains, and therefore is subject to the security policies of those domains.

Security aware applications can administer security policies (providing they are authorized to do so) using the interfaces described in Section 2.4, "Administrator's Interfaces," on page 2-116.

2.3.10 Access Control

2.3.10.1 Description of Facilities

Access policies for applications may be enforced the following ways:

- Automatically by the ORB services on object invocation, to determine whether the caller has the right to invoke an operation on an object.
- By the application itself, to enforce further controls on who can invoke it to do what.
- By the application to control access to its own internal functions and state.

This section is concerned with applications that wish to enforce their own access controls, either supplementing the automatic controls on invocation or controlling internal functions.

As explained in Access Policies under Section 2.1.4, "Access Control Model," on page 2-7, the decision on whether to allow such access may use the following:

- The principal's credentials (which either contain its privilege attributes, or identify the principal so these can be obtained). Using only the principal's identity generally requires that identity to be known at all targets, and leads to scalability problems, so its use is depreciated. Use of the principal's role or group(s) are more likely to give easier administration in large systems, as would security clearance. Enterprise-defined attributes can also be used when supported.
- The target's control attributes such as an ACL or security classification.
- Other relevant information about the action such as the operation (on object invocation) and parameters, and also context information such as time.
  The application can use rights associated with an interface (as described in Section 2.4.3, "Security Policies Introduction," on page 2-118) rather than specify controls for individual operations.
- The security policy rules using this information as enforced by the access decision function.

The access policies enforced automatically by the ORB during object invocation can take into account the principal's credentials, the target's control attributes, the operation and the time (though the time is not used in the standard access policy defined in Section 2.4, "Administrator's Interfaces," on page 2-116). However, the ORB does not use the parameters to the operation for controlling access. So, for example, if there is a rule that only senior managers can authorize expenditure over $5000, the application is likely to need its own function to perform the required check.

Where an application enforces its own access decisions, it will be responsible for maintaining its own control information about operations, functions, and data it wishes to protect. It can do this in a way specific to its own particular functions or data, but in some cases, it is possible to have a more generic way of handling access decisions, and in these cases, it may be possible to use a common access decision object with common administration of the ACLs or other control attributes.

2.3.10.2 *The Access Decision Object*

The access decision functionality is encapsulated in **Access Decision** objects. These may require different information depending on, for example, the action or data to be controlled and the security policy rules as previously described. The **Access Decision** object is a **locality constrained** object.

The **Access Decision** object has the **access_allowed** operation as is used for enforcing access policies in the ORB (see below). The input parameters to this should normally specify:

- The privileges of the initiator of the action. The form of these depends on the specific policy. Some options are:

  - The privileges of the initiator as supplied by a **get_attributes** operation on **Current** (see “The SecurityLevel1::Current Interface” on page 2-95).
  - A credentials object, which represents principal.

- Other information required by the access decision function, including:
  - Application-level decisions on whether an invocation is permitted, the operation and parameters passed in the request, and the object reference.
  - Control of access to internal functions and data, the action, and relevant parameters.

The return value from the **access_allowed** operation is either **TRUE** signifying access is permitted, or **FALSE** signifying that it is not.

It is recommended that where possible, access decisions are made by such **Access Decision** objects (or at least separate internal functions) that hide details of the actual security policy used, so the application does not need to know, for example, whether an ACL or label-based policy is used.

2.3.10.3 The SecurityLevel2::AccessDecision Interface

The **Access Decision** object is a **locality constrained** object. The **AccessDecision** interfaces have the following single operation:

access_allowed

```
boolean access_allowed(
    in      SecurityLevel2::CredentialsList    cred_list,
    in      Object                             target,
    in      CORBA::Identifier                  operation_name,
    in      CORBA::Identifier                  target_interface_name
);
```

Parameters

cred_list

The list of **Credentials** associated with the request. The list may be empty (in the case of unauthenticated requests), it may contain only a single credential, or it may contain several credentials (in the case of delegated or otherwise cascaded requests). The **Access Decision** object is presumed to have rules for dealing with all these cases.

target	The reference used to invoke the target object. The method invoked.
operation_name	The name of the operation being invoked on the target.
target_interface_name	The name of the interface to which the operation being invoked belongs. This may not be required in some implementations and will only be required in cases in which the operation being invoked does not belong to the interface of which the target object is a direct instance.

Return Value

boolean	A return value of **TRUE** indicates that the request should be allowed, otherwise **FALSE .**

2.3.10.4 Portability Implications

Portability of applications enforcing their own access controls is improved by use of **Access Decision** objects as previously described. The application then does not need to know the particular rules used, and even which principal and object attribute types are used to decide whether access should be permitted. It can also hide whether the principal's credentials include all privilege attributes needed, or whether these are obtained dynamically when needed.

Different systems may need to support different access control policies. By hiding details of the access control rules used to enforce the policy behind a standard interface, the application will generally be portable to environments with different policies.

Applications that use their own specific code to make access decisions will only be portable to systems that support the identity and privilege attribute types used in those decisions with the same syntax.

2.3.11 Delegation Facilities

2.3.11.1 Description of Facilities

An operation on a target object may result in calls on many other objects as described in Section 2.1.6, "Delegation," on page 2-13. An intermediate object in this chain of objects may:

- Delegate the credentials received (often containing the initiating principal's privileges) to the next object in the chain, so access decisions at the target may be based on that principal's privileges.
- Act on its own behalf, so use its own credentials when invoking another object in the chain.

- Supply privileges from both, so access decisions at the target object can take into account both the initiating principal's privileges and where these came from.

Which of these delegation modes should be used depends on the application. For example, a user might call a database object asking for some data, and this may obtain the data from a file that also contains data belonging to other users. In this example, the database object would control access to the data using the user's privileges, whereas the filestore object would use the database's privileges.

In general, the delegation mode used is specified by the administrator in the delegation policy for objects of this type in this domain. However, a security aware application can also specify the delegation mode it wants to use, as it may want different modes when invoking different objects.

2.3.11.2 Operations

All the operations used for delegation are specified elsewhere. This section describes how they are used during delegation.

The way the received and intermediate's own credentials are combined in **SecCompositeDelegation** is not defined. Depending on the implementation:

- The initiating principal's and the intermediate's own credentials are passed, and are available separately at the target.
- The received credentials and intermediate's own credentials are combined, so the target sees only a single credentials object with privileges from each of these.
- **Credentials** from all objects in the delegation chain are passed and are available separately to the target.

None of these particular composite delegation modes are part of the Security Functionality Level 2. They are described here because of the effect on the **Current::received_credentials** (see Section 2.3.7.3, "The SecurityLevel2::Current Interface," on page 2-95), which a target object uses to find out who called it. The target normally uses this to get privileges for use in access control decisions.

2.3.11.3 Portability Implications

Where possible, the delegation mode should be set using the administrative interfaces to the delegation policy, so applications may delegate privileges (or not) without any application level code, and so be portable.

If an application sets its own delegation mode, it should be able to handle a CORBA::NO_IMPLEMENT exception if **SecCompositeDelegation** is specified, as this may not be supported.

If the application wants to enforce its own access policy, it should use an **Access Decision** object (as described in Section 2.3.10, "Access Control," on page 2-103), which hides whether access decisions utilize the initiator's privileges separately from the delegate's privileges.

However, where an application wants to provide specific checks which intermediates have been involved in performing the original user's operation, such checks are likely to depend on the delegation scheme and its implementation, and so not be portable.

2.3.12 Non-repudiation

Non-repudiation is an optional facility.

2.3.12.1 Description of Facilities

The Non-repudiation Service provides evidence of application actions in a form that cannot be repudiated later. This evidence is associated with some data (for example, the amount field of a funds transfer document).

Non-repudiation evidence is provided in the form of a token. Two token types are supported:

- Token including the associated data
- Token without included data (but with a unique reference to the associated data)

Non-repudiation tokens may be freely distributed. Any possessor of a non-repudiation token (and the associated data, if not included in the token) can use the non-repudiation Service to verify the evidence. Any holder of a non-repudiation token may store it (along with the associated data, if not included in the token) for later adjudication.

The non-repudiation interfaces support generation and verification of tokens embodying several different types of evidence. It is anticipated that the following will be the most commonly used non-repudiation evidence token types:

- Non-repudiation of Creation prevents a message creator's false denial of creating a message.
- Non-repudiation of Receipt prevents a message recipient's false denial of having received a message.

Generation and verification of non-repudiation tokens require as context a non-repudiation credential, which encapsulates a principal's security information (particularly keys) needed to generate and/or verify the evidence. Most operations provided by the Non-repudiation Service are performed on **NRCredentials** objects.

Non-repudiation Service operations supported by the **NRCredentials** interface are as follows.

- **set_NR_features** specifies the features to apply to future evidence generation and verification operations.
- **get_NR_features** returns the features which will be applied to future evidence generation and verification operations.
- **generate_token** generates a non-repudiation token using the current non-repudiation features. The generated token may contain:
  - Non-repudiation evidence.

  - A request, containing information describing how a partner should use the Non-repudiation Service to generate an evidence token.
  - Both evidence and a request.

- **verify_evidence** verifies the evidence token using the current non-repudiation features.
- **get_token_details** returns information about an input non-repudiation token. The information returned depends upon the type of the token (evidence or request).
- **form_complete_evidence** is used when the evidence token itself does not contain all the data required for its verification, and it is anticipated that some of the data not stored in the token may become unavailable during the interval between generation of the evidence token and verification unless it is stored in the token. The **form_complete_evidence** operation gathers the "missing" information and includes it in the token so that verification can be guaranteed to be possible at any future time.

The **verify_evidence** operation returns an indicator (**evid_complete**), which can be used to determine whether the evidence contained in a token is complete. If a token's evidence is not complete, the token can be passed to **form_complete_evidence** to complete it.

If complete evidence is always required, the call to **form_complete_evidence** can, in some cases, be avoided by setting the **form_complete** request flag on the call to **verify_evidence**; this will result in a complete token being returned via the **evid_out** parameter.

2.3.12.2 Non-repudiation Service Data Types

The following data types are used in the Non-repudiation Service interfaces:

```
module NRservice {
    typedef MechanismType         NRMech;
    typedef ExtensibleFamily      NRPolicyId;

    enum EvidenceType {
        SecProofofCreation,
        SecProofofReceipt,
        SecProofofApproval,
        SecProofofRetrieval,
        SecProofofOrigin,
        SecProofofDelivery,
        SecNoEvidence     // used when request-only token desired
    };

    enum NRVerificationResult {
        SecNRInvalid,
        SecNRValid,
        SecNRConditionallyValid
    };
    typedef unsigned long DurationInMinutes;

    const   DurationInMinutes     DURATION_HOUR = 60;
    const   DurationInMinutes     DURATION_DAY   = 1440;
    const   DurationInMinutes     DURATION_WEEK = 10080;
    const   DurationInMinutes     DURATION_MONTH = 43200;// 30 days
    const   DurationInMinutes     DURATION_YEAR = 525600;//365 days

    typedef long TimeOffsetInMinutes;

    struct NRPolicyFeatures {
        NRPolicyId              policy_id;
        unsigned long           policy_version;
        NRMech                  mechanism;
    };
    typedef sequence <NRPolicyFeatures> NRPolicyFeaturesList;

    // features used when generating requests
    struct RequestFeatures {
        NRPolicyFeatures        requested_policy;
        EvidenceType            requested_evidence;
        string                  requested_evidence_generators;
        string                  requested_evidence_recipients;
        boolean                 include_this_token_in_evidence;
    };
};
```

2.3.12.3 The NRservice::NRCredentials Interface

This section describes the Non-repudiation Service operations that are provided by the **NRCredentials** interface.

set_NR_features

When an **NRCredentials** object is created, it is given a default set of NR features, which determine what NR policy will be applied to evidence generation and verification requests.

Security-aware applications may set NR features to specify policy affecting evidence generation and verification. The interface for setting NR features is:

```
boolean set_NR_features(
    in      NRPolicyFeaturesList          requested_features,
    out     NRPolicyFeaturesList          actual_features
);
```

Parameters

requested_features	The non-repudiation features required.
actual_features	The NR features that were set (may differ from those requested depending on implementation).

Return Value

TRUE	The requested features were equivalent.
FALSE	If the actual features differ from the requested features.

get_NR_features

This operation is provided to allow security-aware applications to determine what NR policy is currently in effect:

```
NRPolicyFeaturesList get_NR_features ();
```

Parameters

None

Return Value

The current set of **NR features** in use in this **NRCredentials** object.

generate_token

This operation generates a non-repudiation token associated with the data passed in an input buffer. Environmental information (for example, the calling principal's name) is drawn from the **NRCredentials** object.

If the data for which non-repudiation evidence is required is larger than can conveniently fit into a single buffer, it is possible to issue multiple calls, passing a portion of the data on each call. Only the last call (i.e., the one on which **input_buffer_complete = true**) will return an output token and (optionally) an evidence check.

```
void generate_token(
    in      Opaque              input_buffer,
    in      EvidenceType        generate_evidence_type,
    in      boolean             include_data_in_token,
    in      boolean             generate_request,
    in      RequestFeatures     request_features,
    in      boolean             input_buffer_complete,
    out     Opaque              nr_token,
    out     Opaque              evidence_check
);
```

Parameters

input_buffer Data for which evidence should be generated.

generate_evidence_type Type of evidence token to generate (may be **SecNoEvidence**).

include_data_in_token If set **TRUE**, data provided in **input_buffer** will be included in generated token; otherwise **FALSE**.

generate_request The output token should include a request, as described in the **request_features** parameter.

request_features A structure describing the request. Its fields are:

- **requested_policy**: Non-repudiation policy to use when generating evidence tokens in response to this request.
- **requested_evidence**: Type of evidence to be generated in response to this request.
- **requested_evidence_generators**: Names of partners who should generate evidence in response to this request.
- **requested_evidence_recipients**: Names of partners to whom evidence generated in response to this request should be sent.
- **include_this_token_in_evidence**: If set true, the evidence token incorporating the request will be included in the data for which partners will generate evidence. If set false, evidence will be generated using only the associated data (and not the token incorporating the request).
- **input_buffer_complete**: True if the contents of the input buffer complete the data for which evidence is to be generated; false if more data will be passed on a subsequent call.
- **nr_token**: The returned NR token.
- **evidence_check**: Data to be used to verify the requested token(s) (if any) when they are received.

Return Value

None.

verify_evidence

Verifies the validity of evidence contained in an input NR token.

If the token containing the evidence to be verified was provided to the calling application by a partner responding to the calling application’s request, then the calling application should pass the evidence check it received when it generated the request as a parameter to **verify_evidence** along with the token it received from the partner.

It is possible to request the generation of complete evidence. This may succeed or fail; if it fails, a subsequent call to **form_complete_evidence** can be made. Output indicators are provided, which give guidance about the time or times at which **form_complete_evidence** should be called; see the parameter descriptions for explanations of these indicators and their use. Note that the time specified by

complete_evidence_before may be earlier than that specified by **complete_evidence_after**; in this case it will be necessary to call **form_complete_evidence** twice.

Because keys can be revoked or declared compromised, the return from **verify_evidence** cannot in all cases be a definitive "**SecNRValid**" or "**SecNRInvalid**"; sometimes "**SecNRConditionallyValid**" may be returned, depending upon the policy in use. "**SecNRConditionallyValid**" will be returned if:

- the interval during which the generator of the evidence may permissibly declare his key invalid has not yet expired (and therefore it is possible that the evidence may be declared invalid in the future), or
- trusted time is required for verification, and the time obtained from the token is not trusted.

```
NRVerificationResult verify_evidence(
    in      Opaque          input_token_buffer,
    in      Opaque          evidence_check,
    in      boolean         form_complete_evidence,
    in      boolean         token_buffer_complete,
    out     Opaque          output_token,
    out     Opaque          data_included_in_token,
    out     boolean         evidence_is_complete,
    out     boolean         trusted_time_used,
    out     TimeT           complete_evidence_before,
    out     TimeT           complete_evidence_after
);
```

Parameters

input_token_buffer	Buffer containing (possibly a portion, possibly all of) evidence token to be verified; buffer may also contain data associated with evidence token (parsing of buffer in this case is understood only by NR mechanism, see **get_token_details)**.
evidence_check	The evidence check.
form_complete_evidence	Set **TRUE** if complete evidence is required; otherwise **FALSE**.
token_buffer_complete	Set **TRUE** if the **input_token_buffer** completes the input token; **FALSE** if more input token data remains to be passed on a subsequent call.
output_token	If **form_complete_evidence** was set to **TRUE**, this parameter will contain complete evidence (and the Return Value will be **SecNRValid**) or an "augmented" but still incomplete evidence token, in which case **SecNRConditionallyValid** is returned.

data_included_in_token	Data associated with the evidence, extracted from input token (may be null).
evidence_is_complete	**TRUE** if evidence in input token is complete, otherwise **FALSE**.
trusted_time_used	**TRUE** if the evidence token contains a time considered to be trusted according to the rules of the non-repudiation policy. **FALSE** indicates that the security policy mandates trusted time and that the time in the token is not considered to be trusted.
complete_evidence_before	If **evidence_is_complete** is **FALSE** and the return value from **verify_evidence** is **SecNRConditionallyValid**, the caller should call **form_complete_evidence** with the returned output token before this time. This may be required, for example, in order to ensure that the time skew between the evidence generation time and the trusted time service's countersignature on the evidence falls within the interval allowed by the current NR policy.
complete_evidence_after	If **evidence_is_complete** is **FALSE** and the return value from **verify_evidence** is **SecNRConditionallyValid**, the caller should call **form_complete_evidence** with the returned output token after this time. This may be required, for example, to ensure that all authorities involved in generating the evidence have passed the last time at which the current NR policy allows them to repudiate their keys.

Return Value

SecNRInvalid	Evidence is invalid.
SecNRValid	Evidence is valid.
SecNRConditionallyValid	Evidence cannot yet be determined to be invalid

get_token_details

The information returned depends upon the type of the token (evidence or request). The mechanism that created the token is always returned.

- If the input token contains evidence, the following is returned: the non-repudiation policy under which the evidence has been generated, the evidence type, the date and time when the evidence was generated, the name of the generator of the evidence, the size of the associated data, and an indicator specifying whether the associated data is included in the token.
- If the input token contains a request, the following is returned: the name of the requester of the evidence, the non-repudiation policy under which the evidence to send back should be generated, the evidence type to send back, the names of the

recipients who should generate and distribute the requested evidence, and the names of the recipients to whom the requested evidence should be sent after it has been generated.

- If the input token contains both evidence and a request, an indicator describing whether the partner's evidence should be generated using only the data in the input token, or using both the data and the evidence in the input token.

```
void get_token_details(
    in      Opaque              token_buffer,
    in      boolean             token_buffer_complete,
    out     string              token_generator_name,
    out     NRPolicyFeatures    policy_features,
    out     EvidenceType        evidence_type,
    out     UtcT                evidence_generation_time,
    out     UtcT                evidence_valid_start_time,
    out     DurationInMinutes   evidence_validity_duration,
    out     boolean             data_included_in_token,
    out     boolean             request_included_in_token,
    out     RequestFeatures     request_features
);
```

Parameters

token_buffer	Evidence token to parse.
token_buffer_complete	Indicator when the token has been fully provided.
token_generator_name	Principal name of token generator.
policy_features	Describes the policy used to generate the token.
evidence_type	Type of evidence contained in the token (may be **SecNoEvidence**).
evidence_generation_time	Time when evidence was generated.
evid_validity_start_time	Beginning of evidence validity interval.
evidence_validity_duration	Length of evidence validity interval.
data_included_in_token	**TRUE** if the token includes the data for which it contains evidence, otherwise **FALSE**.
request_included_in_token	**TRUE** if the token includes a request, otherwise **FALSE**.
evidence_generation_time	Time when evidence was generated.

Return Value

None.

form_complete_evidence

form_complete_evidence is used to generate an evidence token that can be verified successfully with no additional data at any time during its validity period.

```
boolean form_complete_evidence(
    in      Opaque      input_token,
    out     Opaque      output_token,
    out     boolean     trusted_time_used,
    out     TimeT       complete_evidence_before,
    out     TimeT       complete_evidence_after
);
```

Parameters

token_buffer	Evidence token to be completed..
output_token	The “augmented” evidence token may be complete.
trusted_time_used	**TRUE** if the token’s generation time can be trusted, otherwise **FALSE**. If trusted time is required by the policy under which the evidence will be verified, and if this indicator is not set, the evidence will not be considered complete.
complete_evidence_before	If the return value is **FALSE**, **form_complete_evidence** should be called before this time.
complete_evidence_after	If the return value is **FALSE**, **form_complete_evidence** should be called after this time.

Return Value

TRUE	Evidence is now complete.
FALSE	Evidence is not yet complete.

2.4 Administrator’s Interfaces

This section describes the administrative features of the specification. Administration specifies the policies that control the security-related behavior of the system. These features form an ‘Administrator’s View,’ encompassing the interfaces that a human administrator would need to use, but the facilities may also be used by conventional applications that wish to be involved in administrative actions. ‘Administrator’ may therefore refer to a human or system agent.

Most interfaces defined here are in Security Functionality Level 2, as Level 1 security does not include administration interfaces.

2.4.1 Concepts

2.4.1.1 Administrators

This specification imposes no constraints on how responsibilities are divided among security administrators, but in many cases an enterprise will have a security policy that restricts the responsibilities of any one individual. Also, legal requirements may dictate a separation of roles so that, for example, there are different administrators for access control and auditing functions.

Administrators are subject to the same security controls as other users of the system. It is expected that an enterprise will define roles (or other privileges) that certain administrators will adopt. Administrative operations are subject to access controls and auditing in the same way as other object invocations, so only administrators with the required administrative privileges will be able to invoke administrative operations.

This specification does not define administrative functions concerning the management of underlying mechanisms supporting the security services, such as an Authentication Service, Key Distribution Service, or Certification Authority.

2.4.1.2 Policy Domains

Security **administrators** specify security **policies** for particular security policy **domains** (for brevity, only the words in bold are used for the remainder of this section).

A domain includes an object, called the **domain manager**, which has associated with it the policy objects for this domain, and notionally contains zero or more other objects, which are domain **members** and subject to the policies specified by the policy objects associated with the domain manager.

The domain manager records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

There are different types of policy objects for administering different types of policy. As described in "Security Policy Domains" on page 2-21, domains may be members of other domains, forming containment hierarchies. Because different kinds of policy affect different groups of objects, objects (and domains) may be members of multiple domains.

The policies that apply to an object are those of all its enclosing domains.

2.4.1.3 Security Policies

This specification covers administration of security policies, which are enforced by a secure object system in either of the following ways:

- Automatically on object invocation. This covers system policies for security communications between objects, control of whether this client can use this operation on this target object, whether the invocation should be audited, and whether an original principal's credentials can be delegated.
- By the application. This covers security policies enforced by applications. Applications may enforce access, audit, and non-repudiation policies. The application policies may be managed using domains as for other security policies, or the application can choose to manage its own policies in its own way.

Invocation time policies for an object can be applicable only when this object is acting as a client, only when it is a target object, or whenever it is acting as either.

Security policies may be administered by any application with the right to use the security administrative interfaces. This is subject to the invocation access control policy for the administrative interface.

2.4.2 Domain Management

The Domain Management facilities (defined in the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*) are used by the Security Service as described in the following sections.

2.4.3 Security Policies Introduction

Invocation security policies are enforced automatically by ORB services during object invocation. These are:

- **invocation access** policies (**Security::SecClientInvocationAccess** and **Security::SecTargetInvocationAccess,** interface **SecurityAdmin::AccessPolicy**) for controlling access to objects.
- **invocation audit** policies (**Security::SecClientInvocationAudit** and **Security::SecTargetInvocationAudit**, interface **SecurityAdmin::AuditPolicy)** control which operations on which objects are to be audited.
- **invocation delegation** policies (**Security::SecDelegation**, interface **SecurityAdmin::DelegationPolicy**) for controlling the delegation of privileges.
- **secure invocation** policies (**Security::SecClientSecureInvocation** and **Security::SecTargetSecureInvocation**, interface **SecurityAdmin::SecureInvocationPolicy**) for security associations, including controlling the delegation of client's credentials, and message protection.

Different policies generally apply when an object acts as a client from when it is the target of an invocation.

In addition to these invocation policies, there are a number of policy types, which apply independently of object invocation. These are:

- **application access** policy (**Security::SecApplicationAccess**, interface **SecurityAdmin::AccessPolicy**), which applications may use to manage and enforce their access policies.

- **application audit** policy (**Security::SecApplicationAudit**, interface **SecurityAdmin::AuditPolicy**), which applications can use to manage and enforce their audit policies.
- **non-repudiation** policies (**Security::SecNonRepudiation**, interface **SecurityAdmin::NRPolicy**) determine the rules for the generation and use of evidence.

There is also a policy concerned with creation of object references, which is enforced by **POA::create_reference** and variants thereof or equivalent operation. This is the **construction policy** (**CORBA::SecConstruction**) which controls whether a new domain is created when an object of a specified type is created. (See the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*.)

Note – Policies associated with underlying security technology are not included. For example, there are no policies for principal authentication as this is often done by specific security services.

Operations are provided for setting all the types of security policies previously listed. In each case, these management operations permit administration of standard policy semantics supported by the interfaces defined in this specification. It is also possible for implementors to replace the policy objects, the operations of which are defined in this specification, with different policy objects supporting different semantics. In general, such policy objects will also have management operations that are different from those defined in this specification.

2.4.4 Access Policies

There are two types of invocation access policies: 1) the Client Invocation Access policy (**Security::SecClientInvocationAccess**), which is used at the client side of an invocation, and 2) the Target Invocation Access policy (**Security::SecTargetInvocationAccess**), which is used at the target side.

There is one policy type for application access. However, no standard administrative interface to this is specified, as different applications have different requirements.

Access Policies control access by *subjects* (possessing Privilege Attributes), to objects, using *rights*. Privilege Attributes have already been discussed (in Section 2.3, “Application Developer’s Interfaces,” on page 2-71); rights are described in the next section.

2.4.4.1 Rights

The standard **Access Policy** objects in a secure CORBA system implement access policy using *rights* (though implementations may define alternative, non-rights-based **Access Policy** objects).

In rights-based systems, **Access Policy** objects *grant* rights to PrivilegeAttributes. For each operation in the interface of a secure object, some set of rights is *required*. Callers must be granted these required rights in order to be allowed to invoke the operation.

Secure CORBA systems provide a **RequiredRights** interface, which allows:

- Object interface developers to express the "access control types" of their operations using standard *rights,* which are likely to be understood by administrators, without requiring administrators to be aware of the detailed semantics of those operations.
- Access-control checking code to retrieve the rights required to invoke an interface's operations.

A **Required Rights** object is available as an attribute of **Current** in every execution context. Every **Required Rights** object will get and set the same information, so it does not matter which instance of the **RequiredRights** interface is used. The required rights for all operations of all secured interfaces are assumed to be accessible through any instance of **RequiredRights**.

Note that required rights are characteristics of interfaces, *not* of instances. All instances of an interface, therefore, will always have the same required rights.

Note also that because required rights are defined and retrieved through the **RequiredRights** interface, no change to existing object interfaces is required in order to assign required rights to their operations.

Rights Families

This specification provides a standard set of rights for use with the **DomainAccessPolicy** interface defined later in this section. These rights may not satisfy all access control requirements. However; to allow for extensibility, rights are grouped into Rights Families. The **RightsFamily** containing the standard rights is called "**corba**," and contains four rights: "**g**" (interpreted to mean "**get**"), "**s**" (interpreted to mean "**set**"), "**m**" (interpreted to mean "**manage**") and "**u**" (interpreted to mean "**use**"). Implementations may define additional Rights Families. **Rights** are always qualified by the **RightsFamily** to which they belong.

2.4.4.2 *The SecurityLevel2::RequiredRights Interface*

A **Required Rights** object can be thought of as a table (an example Required Rights table appears later in this section). Note that implementations need not manage required rights on an interface-by-interface basis. **Required Rights** objects should be thought of as databases of policy information, in the same way as Interface Repositories are databases of interface information. Thus in many implementations, all calls to the **RequiredRights** interface will be handled by a single Required Rights object instance, or by one of a number of replicated instances of a master Required Rights object instance.

An operation's entry in the Required Rights table lists a set of rights, qualified (or "tagged") as usual with the **RightsFamily**. It also specifies a *Rights Combinator*; the rights combinator defines how entries with more than one required right should be interpreted. This specification defines two Rights Combinators: *AllRights* (which means that all rights in the entry must be granted in order for access to be allowed), and *AnyRight* (which means that if any right in the entry is granted, access will be allowed).

Note that the following behaviors of systems conforming to CORBA Security are unspecified and therefore may be implementation-dependent:

- Assignment of initial required rights to newly created interfaces.
- Inheritance of required rights by newly created derived interfaces.

get_required_rights

This operation retrieves the rights required to execute the operation specified by **operation_name** of the interface specified by **obj. obj**'s interface will be determined and used to retrieve required rights. The returned values are a list of rights and a combinator describing how the list of rights should be interpreted if it contains more than one entry.

```
void get_required_rights(
    in      Object                  obj,
    in      CORBA::Identifier       operation_name,
    in      CORBA::RepositoryId     interface_name,
    out     RightsList              rights,
    out     RightsCombinator        rights_combinator
);
```

Parameters

obj	The object for which required rights are to be returned.
operation_name	The name of the operation for which required rights are to be returned.
interface_name	The name of the interface in which the operation described by **operation_name** is defined, if this is different from the interface of which obj is a direct instance. Not all implementations will require this parameter; consult your implementation documentation If **interface_name** is the empty string, the name of the interface defaults to the most derived interface specified by **obj**.
rights	The returned list of required rights.
rights_combinator	The returned rights combinator.

Return Value

None.

set_required_rights

This operation updates the rights required to execute the operation specified by **operation_name** of the interface specified by **interface_name**. The caller must provide a list of rights and a combinator describing how the list of rights should be interpreted if it contains more than one entry. Note that consistency issues arising from replication of **Required Rights** objects or distribution of the **RequiredRights** interface must be handled correctly by implementations; after a call to **set_required_rights** changes an interface's required rights, all subsequent calls to **get_required_rights**, from any client, must return the updated rights set.

```
void set_required_rights(
    in      CORBA::Identifier         operation_name,
    in      CORBA::RepositoryId       interface_name,
    in      RightsList                rights,
    in      RightsCombinator          rights_combinator
);
```

Parameters

operation_name	The name of the operation for which required rights are to be updated.
interface_name	The name of the interface whose required rights are to be updated.
rights	The desired new list of required rights.
rights_combinator	The desired new **RightsCombinator**.

Return Value

None.

2.4.4.3 The SecurityAdmin::AccessPolicy Interface

This is the root interface for the various kinds of invocation access control policy. This interface supports querying of the effective access granted by a set of attributes by an invocation access policy. It inherits the **CORBA::Policy** interface and has a single operation, **get_effective_rights**.

get_effective_rights

This operation returns the current effective rights (of family **RightsFamily**) granted by this **Access Policy** object to the subject possessing all privilege attributes in the list of attributes **attrib_list**.

```
RightsList get_effective_rights(
    in      AttributeList                     attrib_list,
    in      ExtensibleFamily                  rights_family
);
```

Parameters

attrib_list	A list of attributes obtained from one or more **Credentials** using the **get_attributes** operation.
rights_family	The family of **rights** to be affected, filtering rights that do not that match **rights_family**.

Return Value

A list of effective rights that are consistent with the **attrib_list** and the access policy, of the family specified by **rights_family**. If the rights cannot be mapped from one or more attributes, the attribute is silently ignored.

get_all_effective_rights

This operation returns the current effective rights (for all rights families) granted by this Access Policy object to the subject possessing all privilege attributes in the list of attributes **attrib_list**.

```
RightsList get_all_effective_rights(
    in AttributeList attrib_list
);
```

Parameters

attrib_list	A list of attributes obtained from one or more **Credentials** using the **get_attributes** operation

Return Value

A list of effective rights that are consistent with the **attrib_list** and the **access** policy.

Note that this specification does not define how an **Access Policy** object combines rights granted through different Privilege Attribute entries, in case a subject has more than one Privilege Attribute to which the Access Policy grants rights. However, this call will cause the **Access Policy** object to combine rights granted to all privilege attributes in the input **AttributeList** (using whatever operation it has implemented), and return the result of the combination.

Access Decision objects, and applications that check whether access is permitted without using an **Access Decision** object, should use this operation to retrieve rights granted to subjects.

2.4.4.4 Specific Invocation Access Policies

This specification allows different Invocation Access policies to be provided through specialization of the **AccessPolicy** interface.

The provider of each specific Invocation Access policy is responsible for defining its own administrative operations. This specification defines a standard Invocation Access policy interface, including administrative operations, presented in the next section. This standard policy may of course be replaced by, or augmented with, other policies.

2.4.4.5 The Domain AccessPolicy Object

The **Domain Access Policy** object with the **SecurityAdmin::DomainAccessPolicy** interface provides discretionary access policy management semantics. CORBA implementations with policy requirements, which cannot be met by the **Domain Access Policy** abstraction, may choose to implement different **Access Policy** objects. For example, they may choose to implement access control policy management using capabilities.

Domains

This specification defines interfaces for administration of access policy on a domain basis. Each domain may be assigned an access policy, which is applied to all objects in the domain. Each access-controlled object in a CORBA system must be a member of at least one domain.

A **Domain Access Policy** object defines the access policy, which grants a set of named “subjects” (e.g., users), a specified set of “rights” (e.g., **g**, **s**, **m**, **u**) to perform operations on the “objects” in the domain. A Domain Access Policy can be represented by a table whose row labels are the names of subjects, and whose cells are filled with the rights granted to the subject named in that row’s label, as in Table 2-1. Note that the use of the Delegation State is discussed in “Delegation State” on page 2-125.

Table 2-1 DomainAccessPolicy

Subject	Delegation State	Granted Rights
alice	initiator	corba:gs--
bob	initiator	corba:g---
cathy	initiator	corba:g---
...		
zeke	initiator	corba:gs--

This **Domain Access Policy** grants the rights “**g**” and “**s**” to Alice and Zeke, and the right “**g**” to Bob and Cathy. (The annotation “**corba**” prefixing the granted rights indicates which Rights Family, as defined in the previous section, each of the rights in the table is drawn from. In this case, all rights are drawn from Domain Access Policy’s standard “**corba**” Rights Family. The delegation state column is described under “Delegation State” on page 2-125.

Domain Access Policy Use of Privilege Attributes

Administration of principals by individual identity is costly, so the Domain Access Policy aggregates principals for access control. A common aggregation is called a “user group.” This specification generalizes the way users are aggregated, using “Privilege Attributes” (as defined in Section 2.1.4.3, “Access Policies,” on page 2-9). Users may have many kinds of privilege attributes, including groups, roles, and clearances (note that user access identities, often referred to simply as “user identities” or “userids,” are considered to be a special case of privilege attributes). The **Domain Access Policy** object uses Privilege Attributes as its subject entries.

This specification does not provide an interface for managing user privilege attributes; an implementation of this specification might provide a “User Privilege Attribute Table” enumerating the set of users granted each Privilege attribute. An implementor might provide a user privilege attribute table, shown next.

Table 2-2 User Privilege Attributes (not defined by this specification)

Users	Privilege Attributes
bob, cathy	group:programmers
zeke	group:administrators

Given the definitions in this table, we can simplify our Domain Access Policy as follows (note that, for convenience, each **PrivilegeAttribute** entry is annotated in the table with its **PrivilegeAttribute** type).

Table 2-3 Domain Access Policy (with Privilege Attributes)

Privilege Attribute	Delegation State	Granted Rights
access_id:alice	initiator	corba:gs--
group:programmers	initiator	corba:g---
group:administrators	initiator	corba:gs--

Delegation State

The **Domain Access Policy** abstraction allows administrators to grant different rights when a Privilege attribute is used by a delegate than those granted to the same Privilege attribute when used by an initiator (note that “initiator” means the principal issuing the first call in a delegated call chain; that is, the only client in the call chain that is not also a target object). The **Domain Access Policy** shown next illustrates the use of this feature.

Table 2-4 Domain Access Policy (with Delegate Entry)

Privilege Attribute	Delegation State	Granted Rights
access_id:alice	initiator	corba:gs--

Table 2-4 Domain Access Policy (with Delegate Entry)

access_id:alice	delegate	corba:g---
group:programmers	initiator	corba:g---
group:administrators	initiator	corba:gs--

This **Domain Access Policy** grants Alice the "**g**" and "**s**" rights when she accesses an object as an initiator, but only the "**g**" right when a delegate using her identity accesses the same object.

Domain Access Policy Use of Rights and Rights Families

The rights granted to a Privilege Attribute by a **Domain Access Policy** entry must each be "tagged" with the RightsFamily to which they belong. Each **Domain Access Policy** entry can grant its row's **PrivilegeAttribute** rights from any number of different Rights Families.

Implementations may define new Rights Families in addition to the standard "**corba**" family, though this should be done only if absolutely necessary, since new Rights Families complicate the administrator's model of the system.

Access Decision Use of AccessPolicy and RequiredRights

The **Access Decision** object is described in "The Access Decision Object" on page 2-104. It is used at run-time to perform access control checks. **Access Decision** objects rely upon **Access Policy** objects to provide the policy information upon which their decisions are based.

To complete the example, imagine that we have the following set of object instances..

Table 2-5 Interface Instances

Objects	**Interface**
obj_1, obj_8, obj_n	c1
obj_2, obj_5	c2
obj_12	c3

The **Domain Access Policy** object illustrated next has been updated to include a list of rights of type "other" granted to each of the Privilege attributes..

Table 2-6 Domain Access Policy (with Required Rights Mapping)

Privilege Attribute	**Delegation State**	**Granted Rights**
access_id:alice	initiator	corba: gs-- other: -u-m-s

Table 2-6 Domain Access Policy (with Required Rights Mapping)

access_id:alice	delegate	corba: g--- other: ------
group:programmers	initiator	corba: g--- other: -u----
group:administrators	initiator	corba: gs-- other: ------

Table 2-7 shows Required Rights for three object interfaces (c1, c2, and c3), using the standard Rights Family "**corba**" and a second Rights Family, "other," whose rights set is assumed to be {g, u, o, m, t, s}.

Table 2-7 Required Rights for Interfaces c1, c2, and c3

Required Rights	Rights Combinator	Operation	Interface
corba:s	all	m1	c1
corba:gs	any	m2	
other:u	all	m3	c2
other:ms	all	m4	
other: s	all	m5	c3
corba:gs	all	m6	

Using this, we can calculate the effective access granted by this Domain Access Policy.

- alice can execute operations m1 and m2 of objects obj_1, obj_8, and obj_n as an initiator, but may execute only m2 as a delegate.
- alice can execute operations m3 and m4 of objects obj_2, and obj_5 as an initiator, but may execute no operations of obj_2 and obj_5 as a delegate.
- alice can execute operations m5 and m6 of object obj_12 as an initiator, but may execute no operations as a delegate.
- "programmers" can execute operation m2 of objects obj_1, obj_8, and obj_n as an initiator, but no operations as a delegate.
- "programmers" can execute operation m3 of objects obj_2 and obj_5 as an initiator, but no operations as a delegate.
- "administrators" can execute operations m1 and m2 of objects obj_1, obj_8, and obj_n as an initiator, but no operations as a delegate.
- "administrators" can execute operations m5 and m6 of object obj_12 as an initiator, but no operations as a delegate.

2.4.4.6 *The SecurityAdmin::DomainAccessPolicy Interface*

The **Domain Access Policy** object provides operations for managing access policy through the **DomainAccessPolicy** interface.

Each domain manager may have at most one **Access Policy** object, and therefore at most one **Domain Access Policy** (though an object instance may have more than one domain manager, and therefore, more than one **Domain Access Policy**). The **DomainAccessPolicy** interface inherits the **AccessPolicy** interface and defines operations to specify which subjects can have which rights as follows.

grant_rights

This operation grants the specified `rights` to the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to grant rights to a single privilege attribute.

```
void grant_rights(
    in      Attribute               priv_attr,
    in      DelegationState         del_state,
    in      RightsList              rights
);
```

Parameters

priv_attr	Privilege attributes to be affected.
del_state	Delegation state to be set.
rights	The list of rights to be granted.

Return Value

None.

revoke_rights

This operation revokes the specified **rights** of the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to revoke rights granted to a single privilege attribute.

```
void revoke_rights(
    in      Attribute               priv_attr,
    in      DelegationState         del_state,
    in      RightsList              rights
);
```

Parameters

priv_attr	Privilege attributes to be affected.
del_state	Delegation state to be set.
rights	The list of rights to be revoked.

Return Value

None.

replace_rights

This operation replaces the current rights of the privilege attribute **priv_attr** in delegation state **del_state** with the **rights** provided as input.

Utilities that manage access policy should use this operation to replace rights granted to a single privilege attribute in cases where using **grant_rights** and **revoke_rights** is inappropriate. For example, **replace_rights** might be used to change an **access_id**'s authorizations to reflect a change in job description (since the change in authorization in this case is related to the duties of the new job rather than to the current authorizations granted to the user owning the **access_id**).

```
void replace_rights(
    in      Attribute               priv_attr,
    in      DelegationState         del_state,
    in      RightsList              rights
);
```

Parameters

priv_attr	Privilege attributes to be affected.
del_state	Delegation state to be set.
rights	The list of rights to be replaced.

Return Value

None.

get_rights

This operation returns the current rights (of type **RightsList**) of the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to retrieve rights granted to an individual privilege attribute.

```
RightsList get_rights(
    in      Attribute               priv_attr,
    in      DelegationState         del_state,
    in      ExtensibleFamily        rights_family
);
```

Parameters

priv_attr	Privilege attributes to which the requested rights are granted.
del_state	Delegation state to be set.
rights_family	The family of rights to be affected, filtering rights that do not that match **rights_family**.

Return Value

A list of rights granted to the specified privilege attribute of the specified rights family in the specified delegation state. If the rights cannot be mapped from one or more attributes, the attribute is silently ignored.

get_all_rights

This operation returns the current rights (for all rights families) of the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to retrieve rights granted to an individual privilege attribute.

```
RightsList get_all_rights(
     in     SecAttribute            priv_attr,
     in     DelegationState         del_state
);
```

Parameters

priv_attr	Privilege attributes to which the requested rights are granted.
del_state	Delegation state to be set.

Return Value

A list of rights granted to the specified privilege attribute in the specified delegation state.

2.4.5 Audit Policies

There are two invocation audit policies: 1) the **SecClientInvocationAudit** policy, which is used at the client side of an invocation, and 2) the **SecTargetInvocationAudit** policy, which is used at the target side. There is also an application audit policy type.

Audit policy administration interfaces are used to specify the circumstances under which object invocations and application activities in this domain are audited. As for access policies, this specification allows different audit policies to be specified, which may have different administrative interfaces.

Different audit policies are potentially possible, which allow a great range of options of what to audit. Some of these are needed to respond to the problem of getting the useful information, without generating huge quantities of audit information.

Examples of what events could be audited during invocation include:

- Specified operations on objects.
- Failed operations (i.e., those that raise an exception) on specified object types in a domain.
- Use of certain operations during certain time intervals (e.g., overnight).
- Access control failures on specified operations.
- Operations done by a specified principal.
- Combinations of these.

Note that many of these events may be related to the business application. For example, an operation of **update_bank_account** is a business, rather than system, operation. However, some events are mainly of interest to a Privilege administrator (e.g., access failures to systems objects).

Application audit policies may audit similar types of events, though these are often related to application functions, not object invocations.

2.4.5.1 The SecurityAdmin::AuditPolicy Interface

The **AuditPolicy** interface can be used to administer both client and target invocation audit policies.

This standard audit policy is used to specify, for a set of event families and event types, the selectors to be used to define which events are to be audited.

These are related to the selectors used in **audit_needed** (of **Audit Decision** object, interface **AuditDecision**) and **audit_write** (of **Audit Channel** object, interface **AuditChannel**), as follows..

Table 2-8 Standard Audit Policy

Selector Type	Value on audit_needed and audit_write	Value Administered
InterfaceName	interface name	CORBA::RepositoryId
ObjectRef	object reference	none - the policy applies to all objects in the domain
Operation	op_name	operation
Initiator	credential list	security attributes (audit_id and privileges)
Success Failure	boolean	boolean
Time	utc when event occurred	time interval during which auditing is needed
DayOfWeek	DayOfTheWeek	day of the week on which audit is to be done

Note that audit policy is managed on an audit policy domain basis. Assignment of initial audit selectors to newly created domains is unspecified and hence may be implementation-dependent.

The audit policy also specifies an Audit Combinator for each event type. The Audit Combinator defines how, for a given event type, **audit_needed** matches its selector value list against the selectors in an audit policy. This specification defines two Audit Combinators: **SecAllSelectors** (which means that if all selectors in an audit policy match the selectors supplied to **audit_needed**, **audit_needed** will return **TRUE**), and **SecAnySelector** (which means that if any selector in the audit policy matches a selector in **audit_needed**, **audit_needed** will return **TRUE**).

The following operations are available on the Audit Policy object.

set_audit_selectors

This operation defines the selectors to be used to decide whether to audit the specified event families and types.

```
void set_audit_selectors(
    in      CORBA::RepositoryId     object_type,
    in      AuditEventTypeList      events,
    in      SelectorValueList       selectors,
    in      AuditCombinator         audit_combinator
);
```

Parameters

object_type The type of objects for which an audit policy is being set. If this is the empty string, the default policy for all object types is implied.

events Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.

selectors The values for the selectors to be set for the specified **events**. **Selectors** replaces the old selector list for each of the specified events. (Selectors for all other events remain unchanged.)

audit_combinator The value for the combinator to be set for the specified **events**.

Return Value

None.

clear_audit_selectors

This clears all audit selectors for the specified event families and types.

```
void clear_audit_selectors(
    in      CORBA::RepositoryId     object_type,
    in      AuditEventTypeList      events
);
```

Parameters

object_type The type of objects for which an audit policy is being cleared. If this is the empty string, the default policy for all object types is implied.

events Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.

Return Value

None.

replace_audit_selectors

This replaces the specified selectors.

```
void replace_audit_selectors(
    in      CORBA::RepositoryId     object_type,
    in      AuditEventTypeList      events,
    in      SelectorValueList       selectors
    in      AuditCombinator         audit_combinator
);
```

Parameters

object_type	The type of objects for which an audit policy is being replaced. If this is the empty string, the default policy for all object types is implied.
events	Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.
selectors	The values for the selectors to be set for the specified **events**. **Selectors** replaces the old selector list for each of the specified **events**. **Selectors** for all **events** not in the specified events list are reset to empty lists.
audit_combinator	The value for the combinator to be set for the specified **events**.

Return Value

None.

get_audit_selectors

This obtains the current values of the selectors for the specified event family or event.

```
void get_audit_selectors(
    in      CORBA::RepositoryId     object_type,
    in      AuditEventType          event_type
    out     SelectorValueList       selectors
    out     AuditCombinator         audit_combinator
);
```

Parameters

object_type	The type of objects for which an audit policy is being obtained. If this is the empty string, the default policy for all object types is implied.
eventtype	The requested event type.
selectors	The list of selector values for the specified **event_type**.
audit_combinator	The audit combinator for the specified **event_type**.

Return Value

None.

set_audit_channel

This specifies the identity of the audit channel to be used with this audit policy. The actual audit channel object corresponding to this id is provided to the user by the corresponding Audit Decision object.

```
void set_audit_channel(
    in        AuditChannelId    audit_channel_id
);
```

Parameters

audit_channel_id	A unique identifier associated with an audit channel.

Return Value

None.

2.4.6 Secure Invocation and Delegation Policies

These policies affect the way secure communications between client and target are set up, and then used. There are three policies here:

1. **Security::SecClientSecureInvocation** policy, which specifies the client policy in terms of trust in the target's identity and protection requirements of the communications between them.
2. **Security::SecTargetSecureInvocation** policy, which specifies the target policy in terms of trust in the client's identity and protection requirements of the communications between them.
3. **Security::SecDelegation** policy, which specifies whether credentials are delegated for use by the target when a security association is established between client and target. This is a client side policy.

In all these cases, there is a standard policy interface for administering the policy options. Unlike access and audit policies, this is not replaceable. The standard policy administration operations allow support of a range of policies.

2.4.6.1 Secure Invocation Policies

These are used to set client and target invocation policies which specify both a set of required secure association options and a set of supported options that control how:

- The security association is made, for example, whether trust between client and target is established (implying authentication if the client and target are not in the same identity domain).

- Messages using that association are protected, for example, the levels of integrity and confidentiality.

The administrator should specify the required association options, but will often not need to specify the supported options as these default to the ones supported by the security mechanism used. However, the administrator could choose to restrict what is supported, and in this case, should specify supported options.

Some implementations may support separate sets of association options for communications in the request direction and the reply direction (e.g., for an application that requires no protection on the request, but confidentiality on the reply). Conforming implementations are not required to support this unidirectional feature. Some selectable policy options may not be meaningful to set for a certain direction (e.g., the **EstablishTrustInTarget** option is not meaningful for a reply).

Both **SecClientSecureInvocation** and **SecTargetSecureInvocation** type policy objects support the same interface, though not all of the selectable policy options are meaningful to both client and target.

Required and Supported Secure Invocation Policy

For both the **SecClientSecureInvocation** and **SecTargetSecureInvocation** policies, a separate set of secure association options may be established to indicate **required** policy and **supported** policy. The **required** policy indicates the options that an object requires for communications with a peer. The **supported** policy specifies the options that an object can support if requested by a communicating peer.

The **required** options indicate the minimum requirements of the object, stronger protection is not precluded.

2.4.6.2 Secure Association Options

The selectable secure association options (**Security::AssociationOptions**) are listed next with a description of their semantics for **required** policy and **supported** policy.

NoProtection

- Required semantics: the object's minimal protection requirement is unprotected invocations.
- Supported semantics: the object supports unprotected invocations.

Integrity

- Required semantics: the object requires at least integrity-protected invocations.
- Supported semantics: the object supports integrity-protected invocations.

Confidentiality

- Required semantics: the object requires at least confidentiality-protected invocations.
- Supported semantics: the object supports confidentiality-protected invocations.

DetectReplay

- Required semantics: the object requires replay detection on invocation messages.
- Supported semantics: the object supports replay detection on invocation messages.

DetectMisordering

- Required semantics: the object requires sequence error detection on fragments of invocation messages.
- Supported semantics: the object supports sequence error detection on fragments of invocation messages.

EstablishTrustInTarget

- Required semantics: On client policy, the client requires the target to authenticate its identity to the client. On target policy, this option is not meaningful.
- Supported semantics: On client policy, the client supports having the target authenticate its identity to the client. On target policy, the target is prepared to authenticate its identity to the client.

EstablishTrustInClient

- Required semantics: On client policy, this option is not meaningful. On target policy, the target requires the client to authenticate its privileges to the target.
- Supported semantics: On client policy, the client is prepared to authenticate its privileges to the target. On target policy, the target supports having the client authenticate its privileges to the target.

Note that on an invocation, if both the client and target policies specify that peer trust is needed, mutual authentication of client and target is generally required.

If the target accepts unauthenticated users as well as authenticated ones, the **EstablishTrustInClient** option may be set for **supported** policy, but not for **required** policy. This allows unauthenticated clients to use this target (subject to access controls); the target can still insist on only authenticated users for certain operations by using access controls.

2.4.6.3 The SecurityAdmin::SecureInvocationPolicy Interface

The **SecureInvocationPolicy** interface provides the following operations:

set_association_options

This operations of the **SecurityAdmin::SecureInvocationPolicy** interface (PolicyType **SecClientSecureInvocation** and **SecTargetSecureInvocation**) is used to set the secure association options for objects in the domain to which the policy applies. Separate options may be set for particular object types by using the **object_type** parameter.

This call allows requesting a different set of association options for communication in the request direction versus the reply direction, although conforming implementations are not required to support this feature. The “**request**” and “**reply**” options sets are treated as overrides to the “**both**” options set when evaluating policy for a single communication direction. Implementations should raise the CORBA::BAD_PARAM exception if an unsupported direction is requested on this call.

Not all selectable association options are meaningful for every policy set. For example, **EstablishTrustInClient**, which is meaningful for the **SecTargetSecureInvocation** policy, is not meaningful as a requirement for the **SecClientSecureInvocation** policy. Likewise, certain association options do not make sense when applied to only a single direction (e.g., **EstablishTrustInTarget** is not meaningful for communication in the reply direction). An implementation may choose whether to raise an exception or silently ignore requests for invalid association options.

```
void set_association_options(
    in      CORBA::RepositoryId         object_type,
    in      RequiresSupports            requires_supports,
    in      CommunicationDirection      direction,
    in      AssociationOptions          options
);
```

Parameters

object_type	The type of objects that the association options apply to. If this is nil, all object types are implied
requires_supports	Indicates whether the operation applies to the required options or the supported options
direction	Indicates whether the options apply to only the request, only the reply, or to both directions of the invocation.
options	Indicates requested secure association options by setting the corresponding options flags.

Return Value

None.

get_association_options

This is used to find what secure association options apply on **SecClientSecureInvocation** and **SecTargetSecureInvocation** policy objects for the required or supported policy, for the indicated direction, and for the specified object type.

Implementations should raise the CORBA::BAD_PARAM exception if an unsupported direction is requested on this call.

```
AssociationOptions get_association_options(
    in      CORBA::RepositoryId         object_type,
    in      RequiresSupports            requires_supports,
    in      CommunicationDirection      direction
);
```

Parameters

object_type	The type of objects that the association options apply to. If this is nil, all object types are implied.
requires_supports	Indicates whether the operation applies to the required options or the supported options.
direction	Indicates whether the options apply to only the request, only the reply, or to both directions of the invocation.

Return Values

The association options flags set for this policy.

2.4.6.4 The SecurityAdmin::DelegationPolicy Interface

The **Delegation Policy** object, which has the **SecurityAdmin::DelegationPolicy** interface, controls which credentials are used when an intermediate object in a chain invokes another object.

set_delegation_mode

The **set_delegation_mode** operation specifies which credentials are delegated by default at an intermediate object in a chain where objects invoke other objects. This default can be overridden by the object at run time.

```
void set_delegation_mode(
    in      CORBA::RepositoryId     object_type,
    in      DelegationMode          mode
);
```

Parameters

object_type	The type of objects to which this delegation policy applies.
mode	The delegation mode. Options are: • **SecDelModeNoDelegation**: The intermediate's own credentials are used for future invocations. • **SecDelModeSimpleDelegation**: The initiating principal credentials are delegated. • **SecDelModeCompositeDelegation**: Both the received credentials and the intermediate object's own credentials are passed (if the underlying security mechanism supports this). The requester's credentials and the intermediate's own credentials may be combined into a single credential, or kept separate, depending on the underlying security mechanism.

Return Value

None.

get_delegation_mode

This returns the delegation mode associated with the object.

```
DelegationMode get_delegation_mode(
    in        CORBA::RepositoryId       object_type
);
```

Parameters

object_type	The type of object for which delegation mode is requested.

Return Value

The delegation mode of the object type specified by the **object_type** parameter.

2.4.7 Non-repudiation Policy Management

This section defines interfaces for management of non-repudiation policy.

Non-repudiation policies define the following:

- Rules for the generation of evidence, such as the trusted third parties which may be involved in evidence generation and the roles in which they may be involved and the duration for which the generated evidence is valid.
- Rules for the verification of evidence, for example, the interval during which a trusted third party may legitimately declare its key to have been compromised or revoked.
- Rules for adjudication, for example, which authorities may be used to adjudicate disputes.

The non-repudiation policy itself may be used by the adjudicator when resolving a dispute. For example, the adjudicator might refer to the non-repudiation policy to determine whether the rules for generation of evidence have been complied with.

For each type of evidence, a policy defines a validity duration and whether trusted time must be used to generate the evidence.

For each non-repudiation mechanism, a policy defines the set of trusted third parties ("authorities"), which may be used by the mechanism. A policy also defines, for each mechanism, the maximum allowable "skew" between the time of generation of evidence and the time of countersignature by a trusted time service; if the interval between these two times is larger than the maximum skew, the time is not considered to be trusted.

For each authority, a policy defines which roles the authority may assume, and a time offset, relative to evidence generation time, which allows computation of the last time at which the authority can legitimately declare its key to have been compromised or revoked. For example, if an authority has a defined **last_revocation_check_offset** of negative one hour, then all revocations taking effect earlier than one hour before the generation of a piece of evidence will render that evidence invalid; no revocation taking place later than one hour before the generation of the evidence will affect the evidence's validity. Note that the **last_revocation_check_offset** is inclusive, in the sense that all revocations occurring up to and including the time defined by **generation_time + offset** are considered effective.

2.4.7.1 Data Types for Non-repudiation Policy Management Interfaces

The following data types are used by the NR policy management operations.

```
module NRservice {

    struct EvidenceDescriptor {
        EvidenceType            evidence_type,
        DurationInMinutes       evidence_validity_duration,
        boolean                 must_use_trusted_time
    };

    typedef sequence <EvidenceDescriptor> EvidenceDescriptorList;

    struct AuthorityDescriptor {
        string                      authority_name,
        string                      authority_role,
        TimeOffsetInMinutes         last_revocation_check_offset
        // may be >0 or <0; add this to evid. gen. time to
        // get latest time at which mech. will check to see
        // if this authority's key has been revoked.
    };

    typedef sequence <AuthorityDescriptor> AuthorityDescriptorList;
    struct MechanismDescriptor {
        NRMech                        mech_type,
        AuthorityDescriptorList       authority_list,
        TimeOffsetInMinutes           max_time_skew
        // max permissible difference between evid. gen. time
        // and time of time service countersignature
        // ignored if trusted time not reqd.
    };

    typedef sequence <MechanismDescriptor> MechanismDescriptorList;
};
```

2.4.7.2 The NRservice::NRPolicy Interface

The **NRPolicy** interface has the **get_NR_policy_info** and **set_NR_policy_info** operations, and like all other **Policy** interfaces it derives from the **CORBA::Policy** interface.

get_NR_policy_info

Returns information from a non-repudiation policy object.

```
void get_NR_policy_info(
    out     ExtensibleFamily            NR_policy_id,
    out     unsigned long               policy_version,
    out     TimeT                       policy_effective_time,
    out     TimeT                       policy_expiry_time,
    out     EvidenceDescriptorList      supported_evidence_types,
    out     MechanismDescriptorList     supported_mechanisms
);
```

Parameters

NR_policy_id	The identifier of this non-repudiation policy.
policy_version	The version number of this non-repudiation policy.
policy_effective_time	The time at which this policy came into effect.
policy_expiry_time	The time at which this policy expires.
supported_evidence_types	The types of evidence that can be generated under this policy.
supported_mechanisms	The non-repudiation mechanisms which can be used to generate and verify evidence under this policy.

Return Value

None.

set_NR_policy_info

Updates non-repudiation policy information.

```
boolean set_NR_policy_info(
    in      MechanismDesciptorList      requested_mechanisms,
    out     MechanismDescriptorList     actual_mechanisms
);
```

Parameters

requested_mechanisms	The non-repudiation mechanisms to be supported under this policy.
actual_mechanisms	The non-repudiation mechanisms now supported under this policy.

Return Value

TRUE	The requested mechanisms were all set.
FALSE	If the actual mechanisms returned differ from those requested.

2.5 Implementor's Security Interfaces

This section addresses Security Service replaceability. This section defines the security service interfaces that allow different security service implementations to be substituted, whether or not the generic ORB service interfaces are supported (see Section 2.5.2, "Implementation-Level Security Object Interfaces," on page 2-149, for details).

Appendix E, "Guidelines for a Trustworthy System" offers additional guidance to implementors of secure ORBs, including a discussion of using protection boundaries to separate components, depending on the level of security required.

The description of security interceptors in Section 2.5.1, "Security Interceptors," on page 2-144 (particularly that in Invocation Time Policies), specifies how client and target side policies and client preferences are used to decide what policy options to enforce. This definition of how the options are used applies whether the ORB conforms to the replaceability options or not. The interceptor facility that this is based on is defined in the Interceptors chapter of the *Common Object Request Broker: Architecture and Specification*.

None of the interfaces defined in this section affect the application and administrator's views described in Section 2.3, "Application Developer's Interfaces," on page 2-71, and Section 2.4, "Administrator's Interfaces," on page 2-116.

2.5.1 Security Interceptors

This section describes the interceptors that can be used for implementing the security services.

The ORB Services replaceability package requires implementation of two security interceptors (see the Interceptors chapter of the *Common Object Request Broker: Architecture and Specification*):

- **Secure Invocation Interceptor**. This is a message-level interceptor. At bind time, this establishes the security context required to support message protection. When processing a request, it is a message-level interceptor that uses cryptographic services to provide message protection and verification. It is able to check and protect messages (requests and replies) for both integrity and confidentiality.
- **Access Control Interceptor.** This is a request-level interceptor, which determines whether an invocation should be permitted. This interceptor also handles auditing of general invocation failures, but not related to denial of access (access-control denial failures are audited within the **Access Decision** object, which is called by this interceptor to check access control).

This specification does not define a separate audit interceptor, as the other interceptors' implementations or the security service implementations call Audit Service interfaces directly if the events for which they are responsible are to be audited.

The security interceptors implement security functionality by calling the replaceable security service objects (defined later in this section) as shown in Figure 2-52.

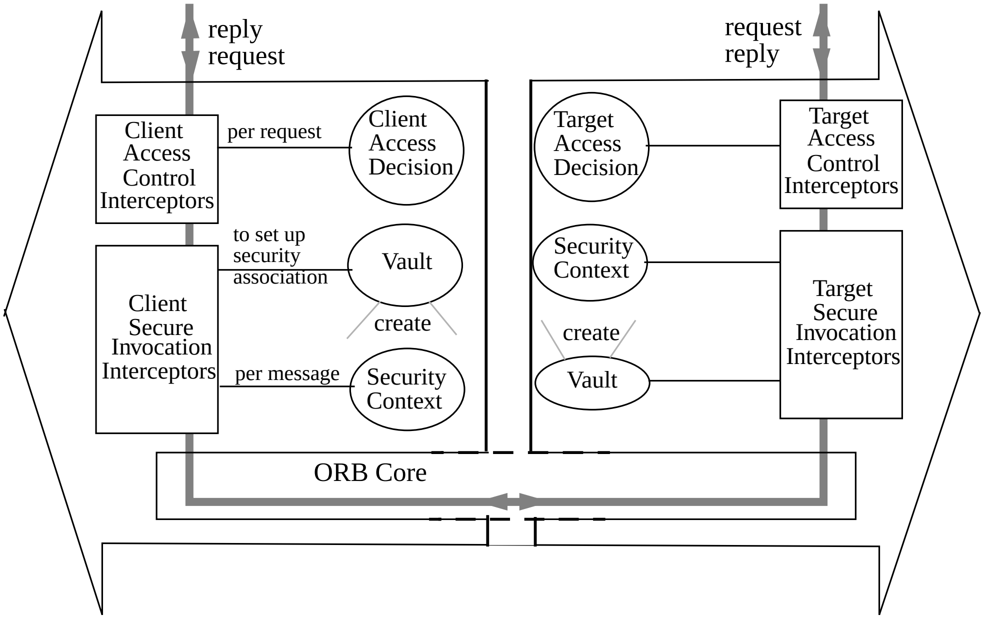

reply
request
request
reply
Client Access Control Interceptors
per request
Client Access Decision
Target Access Decision
Target Access Control Interceptors
to set up security association
Vault
Security Context
create
Client Secure Invocation Interceptors
per message
Security Context
create
Vault
Target Secure Invocation Interceptors
ORB Core

Figure 2-52 Security Functionality Implemented by Security Service Objects

The diagram shows the order in which security interceptors are called. Other interceptors may also be used during the invocation. The order in which other interceptors are called in relationship to security interceptors depends on the type of interceptor.

At the client:

- In general, the access control interceptor should be called first (to avoid unnecessary processing of the request by other interceptors when permission to perform the request is denied).
- All request level interceptors (e.g., transaction or replication ones) are called before the secure invocation interceptor, as the secure invocation interceptor is a message-level interceptor.

  The secure invocation interceptor should ordinarily be the last interceptor invoked (because message protection may encrypt the request, so that the code implementing a further interceptor will not understand it). Even if only integrity protection is used, the integrity check will fail if the message has been altered in any way. Note that data compression and data fragmentation should be applied before the message-protection interceptor is called.

At the target, analogous rules apply to the interceptors in the reverse order.

2.5.1.1 *Invocation Time Policies*

Interceptors decide what security policies to enforce on an invocation as follows:

- They call the **SecurityLevel2::Current::get_policy** operation defined in Section 2.3, “Application Developer’s Interfaces,” on page 2-71, to find what policies apply to this client (at the client side) or this target (at the target side).
- At the client side, the security hints in the target object reference are used to find what policies apply to the target object and what security mechanisms and protocols are supported. This uses operations on the object reference.
- At the client, the overrides set by the client on the credentials or target object reference and the security supported by the mechanism in the client’s environment are taken into account. The Secure Invocation interceptor uses **SecurityLevel2::Current::get_credentials** and **Object::get_policy**.

The **Current::get_policy** operation may be used to get any of the following policies:

- The invocation access policies of the current execution context. These are used by the access control interceptor to check whether access is permitted.
- The invocation audit policy. This is used by interceptors and security services to check whether to audit events during an invocation.
- The secure invocation policy. This is used by the secure invocation interceptor at bind time. It uses **SecureInvocationPolicy::get_association_options** as defined in Section 2.4, “Administrator’s Interfaces. The secure invocation policies (and hints in the object reference) specify required and supported values. The interceptor checks that the required values can be supported, and will not continue

with the invocation if the client's requirements are not met. If the target's requirements are not met, the invocation may optionally proceed, allowing policy enforcement at the target.

- The invocation delegation policy. This is used by the secure invocation interceptor at bind time. The interceptor calls **SecureInvocationPolicy::get_delegation_mode** to retrieve this information.

2.5.1.2 Secure Invocation Interceptor

At bind time, the secure invocation interceptor establishes a security context, which the client initiating the binding can use to securely invoke the target object designated by the object reference used in establishing the binding. At object invocation time, the secure invocation interceptor is called to use the (previously established) security context to protect the message data transmitted from the client to the invoked target object.

Please note that the remainder of this section assumes that security interceptors are implemented using the security services replaceability interfaces defined in this specification; interceptors built for implementations which do not provide the security services replaceability interfaces will have similar responsibilities, but will obviously make different calls.

It should also be noted that binding takes place implicitly and the exact point at which it occurs can vary from one ORB to another. All that one can be certain of is that a binding exists when an invocation of an operation takes place. There is no certainty that the same binding will be used in subsequent invocations. Consequently, the discussion that follows is about binding states and what must happen when the act of implicit binding is executed by the ORB. All reference to the term "Bind" should be interpreted as such.

Bind Time - Client Side

The Secure Invocation interceptor's client bind time functions are used to:

- Find what security policies apply.
- Establish a security association between client and target. This is done on first invoking the object, but may be repeated when changes to the security context occur.

Security policies relevant to this interceptor are the client secure invocation and delegation policies. To retrieve the invocation policy objects, the Secure Invocation interceptor calls the **get_policy** operation.

The interceptor checks if there is already a suitable security context object for this client's use of this target. If a suitable context already exists, it is used. If no suitable context exists, the interceptor establishes a security association between the client and target object (see Section 2.1.3.1, "Establishing Security Associations," on page 2-5).

The client interceptor calls **Vault::init_security_context** to request the security features (such as QOP, delegation) required by the client policy, client overrides and target (as defined in its object reference). The **Vault** returns a security token to be sent to the target, and indicates whether a continuation of the exchange is needed. It also returns a reference to the newly-created **Security Context** object for this client-target security association. (The way trust is established depends on policy, the security technology used, and whether both client and target object are in the same identity domain. It may involve mutual authentication between the objects and negotiation of mechanisms and/or algorithms.)

The interceptor constructs the association establishment message (including the security token, which must be transferred to the target to permit it to establish the target-side **Security Context** object). The association establishment message may be constructed in one of two ways:

1. When only the initial security token is needed to establish the association, the association establishment message may also include the object invocation in the buffer (i.e., the request) supplied to the interceptor when it was invoked by **send_message**. After constructing the association establishment message, the interceptor invokes **send**, which results in the ORB sending the message to the target. After receipt at the target, the association establishment message is intercepted by the Secure Invocation Interceptor in the target, which at bind time calls **Vault::accept_security_context** to create the target **Security Context** object (if needed).

2. When several exchanges are required to establish the security association, the association establishment message is sent separately, in a message that does not include the object invocation in the buffer (i.e., the request), again using **send**. This message is intercepted in the target and the **Vault** called to create the **Security Context** object. However, in this case, the target interceptor must generate another security token and send it back to the client interceptor. The client interceptor calls the **Security Context** object with a **continue_security_context** operation passing the token returned from the target to check if trust has now been established. There may be several exchanges of security tokens to complete this. Once the security association has been established, the original client object invocation (i.e., request) is sent in a separate association establishment message.

Details of the transformation to the request and the association establishment message formats appear in Section 3.1, “Security Interoperability Protocols,” on page 3-1.

Bind Time - Target Side

The secure invocation interceptor’s target bind functions:

- Find the target secure invocation policies.
- Respond to association establishment messages from the client to establish security associations.

On receiving an association establishment message, the target secure invocation interceptor separates it (if needed) into the security token and the request message and uses the **Vault** (if there is no security context object yet) or the appropriate **Security**

Context object to process the security token. As previously described, this may result in exchanges with the client. Once the association is established, the message protection function described next is used to reclaim the request message and protect the reply.

Message Protection (Client and Target Sides)

The Secure Invocation Interceptor is used after bind time for message protection, providing integrity and/or confidentiality protection of requests and responses, according to quality of protection requirements specified for this security association in the active **Security Context** object.

The Secure Invocation Interceptor's **send_message** method calls **SecurityContext::protect_message,** and its **receive_message** method calls **SecurityContext::reclaim_message**, in each case using the appropriate **Security Context** object.

2.5.1.3 Access Control Interceptor

Bind Time

At bind time, the client access control interceptor uses **Current::get_policy** to get the **SecClientInvocationAccess** policy and **SecClientInvocationAudit** policy. The target access control interceptor uses the **get_policy** interface on the **Current** object to get the **SecTargetInvocationAccessPolicy** and **SecTargetInvocationAudit** policy.

Access Decision Time

The Access Control Interceptor decides whether a request should be allowed or disallowed.

Access control decisions may be made at the client side, depending on the client access control policy, and at the target side depending on the target's access control policy. Target side access controls are the norm; client-side access controls can be used to reduce needless network traffic in distributed ORBs. Note that in some ORBs, system integrity considerations may make exclusive reliance on client-side access control enforcement undesirable.

The Access Control Interceptor **client_invoke** and **target_invoke** methods invoke the **access_allowed** method of the **Access Decision** object, specifying the appropriate authorization data. The access decision returns a boolean specifying whether the request should be allowed or disallowed.

The Access Control Interceptor does not know what sort of policy this **Access Decision** object supports. It may be ACL-based, capability-based, label-based, etc. It also does not know if the **Access Decision** object uses the credentials exactly as passed, or takes the identity from the credentials and uses these to find further valid privileges if needed for this principal from a trusted source.

The Access Control Interceptor may also check if this invocation attempt should be audited, by calling the **audit_needed** operation on the **Audit Decision** object; if this call indicates that the invocation attempt should be audited, the Access Control Interceptor uses the **AuditChannel** interface to write the appropriate audit record.

This interceptor does not transform the request. It either passes the request unchanged to continue processing the request, or it aborts the request by returning with an appropriate exception (e.g., CORBA::NO_PERMISSION if **AccessDecision::access_allowed** returns False).

2.5.2 Implementation-Level Security Object Interfaces

The interfaces described in this section are all provided by the underlying security infrastructure and the Object Security Service is a client of these interfaces. Since the interfaces are internal to the ORB Security implementation, all these interfaces are **locality constrained**.

This specification defines the following implementation-level security object interfaces to support security service replaceability:

- **Vault** is used to create a security context for a client/target-object association.
- **Security Context** objects hold security information about the client-target security association and are used to protect messages.
- **Credentials** object is used for passing **Credentials** information between the security infrastructure and the ORB Security Services.
- **Access Decision** objects are used (usually by Access Control Interceptors) to decide if requests should be allowed or disallowed.
- **Audit Decision.** objects are used to decide if events are to be audited.
- **Audit Channel** objects are used to write audit records to the audit trail.
- **Principal Authenticator** object is used for authenticating a principal.
- **NRCredentials** object is used for passing non repudiation credentials informations.

While many of these objects have interfaces that are defined in the context of user interfaces, the Security Replaceability versions of these objects are implemented by the underlying security infrastructure, and the ORB Security Services are the clients of these interfaces.

2.5.2.1 The Vault Object

The Vault object with the **SecurityReplaceable::Vault** interface facilitates creating credentials objects and establishing security contexts between clients and targets when they are in different trust domains. Authentication is required to establish trust. The *Vault* is a **locality constrained** object. Implementations of the *Vault* are responsible for calling **AuditDecision::audit_needed** to determine whether the audit policy requires auditing of successful and/or failed access control checks, and for calling **AuditChannel::audit_write** whenever audit is needed.

2.5.2.2 The SecurityReplaceable::Vault Interface

The **Vault** operations are described below. Note that if an invocation of a **Vault** operation results in an incomplete **Security Context** (i.e., one which requires continued dialogue to complete), the continuation of the dialogue is accomplished using the interface of the incomplete **Security Context** object rather than the **Vault** interface.

acquire_credentials

This operation is called to authenticate the principal and optionally request privilege attributes that the principal requires during its capsule specific session with the system. It creates a capsule specific **Credentials** object including the required attributes.

```
AuthenticationStatus acquire_credentials(
    in      AuthenticationMethod     method,
    in      MechanismType            mechanism,
    in      SecurityName             security_name,
    in      Opaque                   auth_data,
    in      AttributeList            privileges,
    out     Credentials              creds,
    out     Opaque                   continuation_data,
    out     Opaque                   auth_specific_data
);
```

Parameters

method	Contains the identifier of the authentication method used
mechanism	Contains the security mechanism with which to create the **Credentials**.
security_name	Contains the principal’s identification information (e.g., login name).
auth_data	Contains the principal’s authentication information such as password or long term key.
privileges	Contains the privilege attributes requested.
creds	Contains the **locality constrained** object reference of the newly created **Credentials** object. It is usable and placed on the **Current** object’s **own_credentials** list only if the return value is ‘**SecAuthSuccess**.’
auth_specific_data	Information specific to the particular authentication service used
continuation_data	If the return parameter from the authenticate operation is ‘**SecAuthContinue**,’ then this parameter contains challenge information for authentication continuation.

Return Value

The return parameter is used to specify the result of the operation.

'SecAuthSuccess'	Indicates that the object reference of the newly created initialized credentials object is available in the **creds** parameter.
'SecAuthFailure'	Indicates that authentication was in some way inconsistent or erroneous, and therefore credentials have not been created.
'SecAuthContinue'	Indicates that the authentication procedure uses a challenge/response mechanism. The **creds** contains the object reference of a partially initialized **Credentials** object. The **continuation_data** indicates details of the challenge.
'SecAuthExpired'	Indicates that the authentication data contained some information, the validity of which had expired (e.g., expired password). **Credentials** have therefore not been created.

continue_credentials_acquistion

This continues the authentication process for authentication procedures that cannot complete in a single operation. An example of this might be a challenge/response type of authentication procedure.

```
AuthenticationStatus continue_credentials_acquisition(
    in     Opaque                response_data,
    in     Credentials           creds,
    out    Opaque                continuation_data,
    out    Opaque                auth_specific_data
);
```

Parameters

response_data	Contains the response data to the challenge.
creds	Contains the reference of the partially initialized Credentials object. The Credentials object is fully initialized only when return parameter is '**SecAuthSuccess**.'
continuation_data	If the return parameter from the continue_authentication operation is '**SecAuthContinue**,' then this parameter contains challenge information for authentication continuation.
auth_specific_data	Contains information specific to the particular authentication service used.

Return Value

The return parameter is used to specify the result of the operation.

‘SecAuthSuccess’	Indicates that the **Credentials** object whose reference was identified by the **creds** parameter is now fully initialized.
‘SecAuthFailure’	Indicates that the response data was in some way inconsistent or erroneous, and that therefore credentials have not been created.
‘SecAuthContinue’	Indicates that the authentication procedure requires a further challenge/response. The **Credentials** object whose reference was identified in the **creds** parameter is still only partially initialized. The **continuation_data** indicates details of the next challenge.
‘SecAuthExpired’	Indicates that the authentication data contained some information whose validity had expired (e.g., expired password). The **Credentials** object referred to by the **creds** parameter is not valid.

init_security_context

This operation is used by the association interceptor (or the ORB if separate interceptors are not implemented) at the client to initiate the establishment of a security association with the target. This operation creates the **ClientSecurityContext** object that represents the client’s view of the shared security context.

```
AssociationStatus init_security_context(
    in     Credentials                 creds,
    in     SecurityName                target_security_name,
    in     Object                      target,
    in     DelegationMode              delegation_mode,
    in     OptionsDirectionPairList    association_options,
    in     MechanismType               mechanism,
    in     Opaque                      mech_data,
    in     Opaque                      chan_bindings,
    out    OpaqueBuffer                security_token,
    out    ClientSecurityContext       security_context
);
```

Parameters

creds The credentials to be used to establish the security association.

target_security_name The security name of the target as set in its object reference.

target The target object reference.

delegation_mode The mode of delegation to employ. The value is obtained by combining client policy and application preferences as described in Invocation Time Policies under Section 2.5.1, “Security Interceptors,” on page 2-144.

association_options A sequence of one or more pairs of secure association options and direction. The options include such things as required peer trust and message protection. Normally, one pair will be specified, for the “both” direction. Implementations that support separate association options for requests and replies may supply an additional options set for each direction supported. These values are obtained from a combination of the client’s security policy, the hints in the target object reference, and any requests made by the application.

mechanism Normally **NULL**, meaning use default mechanism for security associations. Otherwise, it contains the security mechanism(s) requested. (These may have been obtained from the target object reference.)

mech_data Any data specific to the chosen mechanism, as found in the target object reference

chan_binding Normally NULL (zero length). If present, they are channel bindings as in GSS-API.

security_token The token to be transmitted to the target to establish the security association. Note that this may take several exchanges, but operations required at the client to continue the establishment of the association are on the **Security Context** object

security_context The initialized security context.

Return Value

The return value is used to specify the result of the operation.

SecAssocSuccess	Indicates that the security context has been successfully created and that no further interactions with it are needed to establish the security association.
SecAssocFailure	Indicates that there was some error, which prevents establishment of the association.
SecAssocContinue	Indicates that the association procedure needs more exchanges.

accept_security_context

This operation is used by the association interceptor (or ORB) at the target to accept a request from the client to establish a security association. This operation creates the **ServerSecurityContext** object that represents the target's view of the shared security context.

```
AssociationStatus accept_security_context(
     in      CredentialsList              creds_list,
     in      Opaque                       chan_bindings,
     in      OpaqueBuffer                 in_token,
     out     OpaqueBuffer                 out_token,
     out     ServerSecurityContext        security_context
);
```

Parameters

creds_list	The credentials of the target. Note that this may be the credentials of the trust domain, not the individual object.
chan_bindings	If present, the channel bindings are as in GSS-API.
in_token	The security token transmitted from the client.
out_token	If establishment of the security association is not yet complete, this contains the security token to be transmitted to the client to continue the security dialogue. Note that any further operations needed to complete the security association are on the security context object.
security_context	The **Security Context** object at the target which represents the shared security context between client and target.

Return Value

SecAssocSuccess	Indicates that the security context has been successfully created and no further interactions with it are needed to establish the security association.
SecAssocFailure	Indicates that there was some error that prevents establishment of the association.
SecAssocContinue	The first stage of establishing the security association has been successful, but it is not complete. The **out_token** contains the token to be returned to continue it.

get_supported_mechs

This operation returns the mechanism types supported by this Vault object and the association options these support.

```
MechandOptionsList get_supported_mechs ();
```

Parameters

None.

Return Value

The list of mechanism types supported by this Vault object and the association options they support.

get_supported_authen_methods

This operation returns the authentication methods that are valid for a particular mechanism that the Vault object supports. This operation raises a CORBA::BAD_PARAM exception if the vault does not support the mechanism.

```
AuthenticationMethodList get_supported_authen_methods(
    in        MechanismType             mechanism
);
```

Parameters

mechanism	Contains the mechanism for which the authentication methods are valid.

Return Value

The list of authentication methods supported by this Vault object for the particular mechanism.

2.5.2.3 The Security Context Object

A **Security Context** object with the **SecurityReplaceable::SecurityContext** interface represents the shared security context between a client and a target. It is a **locality constrained** object. It is used as follows:

- By the security association interceptors to complete the establishment of a security association between client and target after the **Vault** has initiated this.
- By the message protection interceptors in protecting messages for integrity and/or confidentiality.
- In response to a target object's request to **Current** for privileges and other information (sent from the client) about the initiating principal.
- In response to a target object's request to **Current** to supply one (or more) credentials object(s) from incoming information about principal(s).
- To check if the security context is valid, and if not, try and refresh it.

The Security Context object is a stateful object that goes through state transitions based on the result of calls on its operations. It also may go through state transitions based on environmental concerns such as an amount of time that has expired. An implementation of a Security Context must model the following states:

Initial	Initial state of any Security Context.
Continued	The Security Context is in process of negotiation and not yet established. This state corresponds to SECIOP state S1 and S3.
ClientEstablished	The Security Context is established on the client side. This means evidence from the target may not need to be processed before messages can be protected and sent to the target side. This state corresponds to SECIOP state S2.
Established	The Security Context is fully established. It is able to process all messages. This state corresponds to SECIOP state S3.
EstablishExpired	The negotiation has expired.
Expired	The Security Context has expired.
Invalid	The Security Context is invalid.

The state transitions are modeled by the following diagram

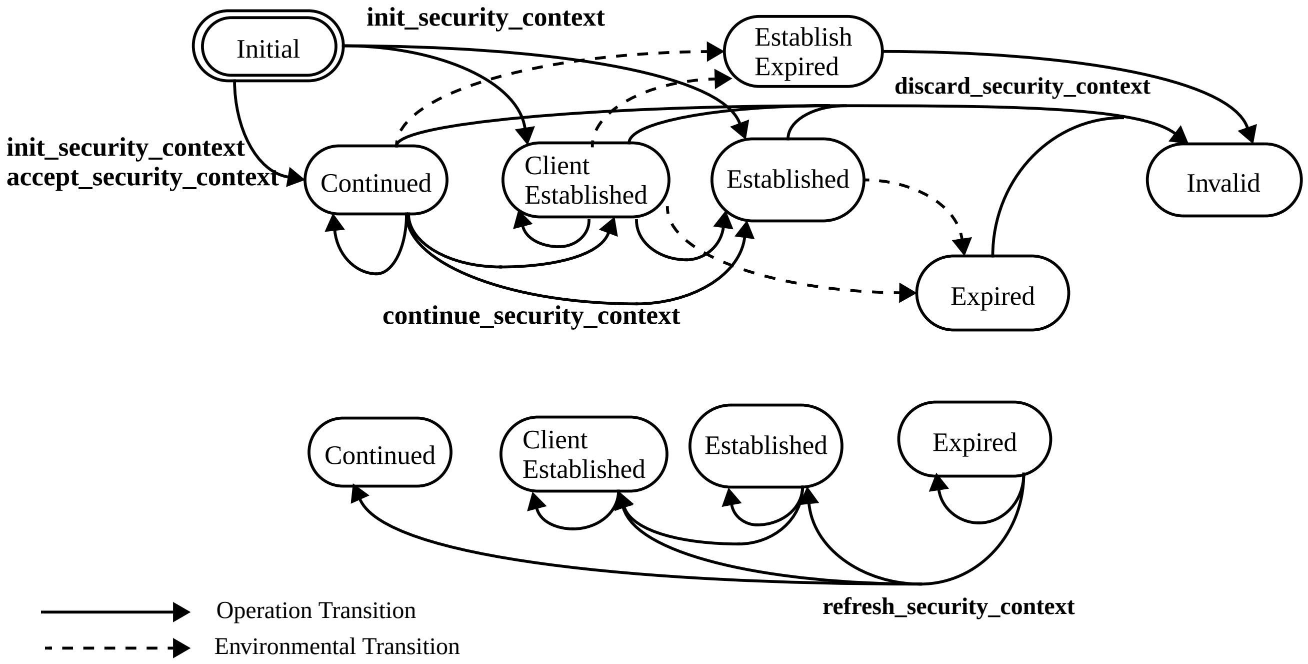


Figure 2-53 Security Context State Transition Diagram

An implementation of a Security Context that transitions into the *ClientEstablished* state, which must only be on the client side of the context, must allow successful processing of **protect_message** operations.

From any state, a context may enter the *Expired* or *Invalid* (not pictured) states due to environmental events or bad operations. Contexts in the *ClientEstablished*, *Established*, and *Expired* state may be refreshed, although, it is not a requirement that refresh be successful for all those states (i.e. some mechanisms may only allow refresh of unexpired contexts). If refresh is not supported for this context, then the **supports_refresh** attribute must be false.

2.5.2.4 The SecurityReplaceable::SecurityContext Interface

The **SecurityReplaceable::SecurityContext** interface has the following attributes and operations:

context_type

The **context_type** readonly attribute returns the orientation type of the security association. It has the following definition:

readonly attribute SecurityContextType context_type;

Return Value

'SecClientSecurityContext'	This security context has a client orientation. It was created by the **Vault::init_security_context** operation.
'SecServerSecurityContext'	This security context has a server orientation. It was created by the **Vault::accept_security_context** operation.

context_state

The **context_state** readonly attribute returns state of the security association. A security context goes through a number of different states during the establishment and use of the secure association. It has the following definition:

readonly attribute SecurityContextState context_state;

Return Value

'SecContextInitialized'	This security context has been initialized.
'SecContextContinued'	This security context is awaiting more negotiation to become established.
'SecContextClientEstablished'	This security context is established on the client side and the client has the ability to send protected messages to the server side. However, the context is still waiting for the server side to complete the establishment of the association.
'SecContextEstablished'	This security context is fully established.
'SecContextEstablishExpired'	This security context has expired during establishment negotiation.
'SecContextExpired'	This security context has expired. It may be possible to refresh it.
'SecContextInvalid'	This security context is invalid. It cannot be used or refreshed.

mechanism

The **mechanism** readonly attribute returns security mechanism used by security association. It has the following definition:

readonly attribute MechanismType mechanism;

Return Value

The value of the mechanism that created the security context.

supports_refresh

The **supports_refresh** readonly attribute returns whether the mechanism and the implementation of this **SecurityReplaceable::SecurityContext** object can support refreshment of the security context.

```
readonly attribute boolean supports_refresh;
```

Return Value

FALSE	Refresh is not supported.
TRUE	Refresh is supported.

chan_binding

The **chan_binding** readonly attribute returns channel binding that was used when the security context was created. It has the following definition:

```
readonly attribute Opaque chan_binding;
```

Return Value

The channel binding that was used when the security context was created.

received_credentials

The **received_credentials** readonly attribute returns the **ReceivedCredentials** that are received from the invoker.

```
readonly attribute ReceivedCredentials received_credentials;
```

Return Value

Object reference to received credentials.

continue_security_context

This operation is invoked by the association interceptor to continue establishment of the security association. It can be called by either the client or target interceptor on the local security context object.

```
AssociationStatus continue_security_context(
    in      OpaqueBuffer        in_token,
    out     OpaqueBuffer        out_token
);
```

Parameters

in_token	The security token generated by the other one of the client-target pair and sent to this **Security Context** object to be used to continue the dialogue between client and target to establish the security association.
out_token	If required, a further security token to be returned to the other **Security Context** object to continue the dialogue.

Return Value

SecAssocSuccess	The security association has been successfully established.
SecAssocFailure	The attempt to establish a security association has failed.
SecAssocContinue	The context is only partially initialized and further operations are required to complete authentication.

protect_message

The **protect_message** operation of the **Security Context** object provides the means whereby the client message protection interceptor may protect the request message, or the target interceptor may protect the response message for integrity and/or confidentiality according to the Quality of Protection required.

```
void protect_message(
    in      OpaqueBuffer        message,
    in      QOP                 qop,
    out     OpaqueBuffer        text_buffer,
    out     OpaqueBuffer        token
);
```

Parameters

message	The message for which protection is required.
qop	Required message protection options.
text_buffer	The protected message, optionally encrypted.
token	The integrity checksum, if any.

Return Value

None.

reclaim_message

The **reclaim_message** operation on the **SecurityContext** object provides the means whereby a protected message may be checked for integrity and decrypted if necessary.

```
boolean reclaim_message(
    in      OpaqueBuffer        text_buffer,
    in      OpaqueBuffer        token,
    out     QOP                 qop,
    out     OpaqueBuffer        message
);
```

Parameters

text_buffer	The message for which the check is required and optionally the message to be decrypted.
token	The integrity checksum, if any. Will typically be zero length if QOP indicates that confidentiality was applied.
qop	The quality of protection that was applied to the protected message.
message	The unprotected message, decrypted if required.

Return Value

If the **reclaim_message** operation returns a value of **FALSE**, then the message has failed its integrity check. If **TRUE**, the integrity of the message can be assured.

is_valid

The **is_valid** operation of the **Security Context** object allows a caller to determine whether the context is currently valid.

```
boolean is_valid(
    out     UtcT                expiry_time
);
```

Parameters

expiry_time	The time at which this context is no longer valid.

Return Value

FALSE	The context is no longer valid.
TRUE	The context is still valid.

refresh_security_context

This operation may extend the useful lifetime of the **SecurityContext**. It takes one input argument of data specific to the mechanism that may be needed to complete a refresh of the context. The output token should be given as evidence to the opposite side of the refresh. The **refresh_security_context** operation may be called on both valid and expired contexts.

Note – Refreshing a security context may possibly reopen the context for possible renegotiation of the security context. Implementations should check the state of the security context to determine if calls to **continue_security_context** may be needed to complete refreshment of the security context.

```
boolean refresh_security_context (
    in      Opaque                 refresh_data,
    out     OpaqueBuffer           out_token
);
```

Parameters

refresh_data	Data specific to the mechanism that may be needed to refresh the security context.
out_token	Evidence of the refresh request that is to be delivered to the opposite side of the context.

Return Value

FALSE	The context has not been successfully refreshed. The parameter **out_token** does not contain a valid value.
TRUE	The context has been successfully refreshed, or it has been opened up for renegotiation that may need subsequent calls to **continue_security_context**. The parameter **out_token** contains the evidence token.

process_refresh_token

This operation may extend the useful lifetime of the **SecurityContext** based on a token from the opposite side of the shared association. The **refresh_security_context** operation may be called on both valid and expired contexts provided that they have not yet been destroyed or discarded.

Note – Refreshing a security context may possibly reopen the context for possible renegotiation of the security context. Implementations should check the state of the security context to determine if calls to **continue_security_context** may be needed to complete refreshment of the security context.

```
boolean process_refresh_token (
    in      OpaqueBuffer           refresh_token,
);
```

Parameters

refresh_token	Evidence token supporting refresh of this context.

Return Value

FALSE	The context has not been successfully refreshed.
TRUE	The context has been successfully refreshed, or it has been opened up for renegotiation that may need subsequent calls to **continue_security_context**.

discard_security_context

This operation is invoked by the association interceptor to discard a security association. It takes one input argument of data specific to the mechanism that may be needed to discard the context. The output token may be given as evidence to the opposite side of the discard.

```
boolean discard_security_context (
    in      Opaque              discard_data,
    out     OpaqueBuffer        out_token
);
```

Parameters

refresh_data	Data specific to the mechanism that may be needed to discard the security context.
out_token	Evidence of the discard to be delivered to the opposite side.

Return Value

FALSE	The context has not been discarded. The parameter **out_token** does not have a valid value.
TRUE	The context has been discarded. The parameter **out_token** contains the evidence token.

process_discard_token

This operation may discard the **SecurityContext** based on a token from the opposite side of the shared association. The **process_discard_token** operation may be called on both valid and expired contexts.

```
boolean process_discard_token (
    in      OpaqueBuffer        discard_token,
);
```

Parameters

discard_token	Evidence token supporting discard of this context.

Return Value

FALSE	The context has not been discarded. Discard token may be invalid for context.
TRUE	The context has been successfully discarded.

2.5.2.5 *The Client Security Context Object*

A **Client Security Context** object with the **SecurityReplaceable::ClientSecurityContext** interface represents the client's view of a shared security context between a client and a target. It implements the **SecurityReplaceable::SecurityContext** interface by inheritance and is a **locality constrained** object.

2.5.2.6 *The SecurityReplaceable::ClientSecurityContext Interface*

The **SecurityReplaceable::ClientSecurityContext** interface extends the **SecurityReplaceable::SecurityContext** interface with attributes that concern client side initialization arguments and target side information. It has the following attributes:

association_options_used

The **asscociation_options_used** readonly attribute returns the association options used and to create the security context with **Vault::init_security_context**. These options may also have been negotiated during set up to something other than the association options supplied to **Vault::init_security_context**. Nonetheless, it is the current state of the security context that is reflected in this attribute.

```
readonly attribute AssociationOptions association_options_used;
```

Return Value

The association options that reflects the current state of the security context.

delegation_mode

The **delegation** readonly attribute returns the delegation mode used and to create the security context with **Vault::init_security_context**. This option may have been negotiated during set up to something other than the association options supplied to **Vault::init_security_context**. Nonetheless, it is the delegation mode of the security context that is reflected in this attribute.

readonly attribute Security::DelegationMode delegation_mode;

Return Value

The delegation mode that reflects the current state of the security context.

mech_data

The **mech_data** readonly attribute returns the value of the **mech_data** argument used to create the security context with **Vault::init_security_context**.

readonly attribute Opaque mech_data;

Return Value

The mechanism data used to create the context.

client_credentials

The **client_credentials** readonly attribute returns the credentials used and to create the security context with **Vault::init_security_context**.

readonly attribute Credential client_credentials;

Return Value

The credentials used to create the security context.

server_options_supported

The **server_options_supported** readonly attribute returns the association options that the server side of the security context supported.

readonly attribute AssociationOptions server_options_supported;

Return Value

The association options that the server supports.

server_options_required

The **server_options_required** readonly attribute returns the association options that the server side of the security context required.

readonly attribute AssociationOptions server_options_required;

Return Value

The association options that the server requires.

server_security_name

The **server_security_name** readonly attribute returns the security name that the server side of the security context represents.

readonly attribute Opaque server_security_name;

Return Value

The security name of the target side.

2.5.2.7 The Server Security Context Object

A **Server Security Context** object with the **SecurityReplaceable::ServerSecurityContext** interface represents the target's view of a shared security context between a client and a target. It implements the **SecurityReplaceable::SecurityContext** interface by inheritance and is a **locality constrained** object.

2.5.2.8 The SecurityReplaceable::ServerSecurityContext Interface

The **SecurityReplaceable::ServerSecurityContext** interface extends the **SecurityReplaceable::SecurityContext** interface with attributes that concern target side initialization arguments and target side information. It has the following attributes:

association_options_used

The **asscociation_options_used** readonly attribute returns the association options that have been negotiated during set up via **Vault::accept_security_context**.

readonly attribute AssociationOptions association_options_used;

Return Value

The association options that reflects the current state of the security context.

delegation_mode

The **delegation** readonly attribute returns the delegation mode in effect for this security context.

readonly attribute Security::DelegationMode delegation_mode;

Return Value

The delegation mode that reflects the current state of the security context.

server_credentials

The **server_credentials** readonly attribute returns the server credentials selected from the list of credentials used to create the security context with **Vault::accept_security_context**.

readonly attribute Credentials server_credentials;

Return Value

The credentials used to create the security context.

server_options_supported

The **server_options_supported** readonly attribute returns the association options that this server side of the security context supported.

readonly attribute AssociationOptions server_options_supported;

Return Value

The association options that this server supported for negotiation of this security context.

server_options_required

The **server_options_required** readonly attribute returns the association options that this server side of the security context required.

readonly attribute AssociationOptions server_options_required;

Return Value

The association options that this server required for negotiation of this security context.

server_security_name

The **server_security_name** readonly attribute returns the security name for which this server used to accept and negotiate the security context.

readonly attribute Opaque server_security_name;

Return Value

The target security name of the security context.

2.5.2.9 The Credentials Object

The **Credentials** object with the **SecurityLevel2::Credentials** interface, as defined in Section 2.3.4, “The Credentials Object,” on page 2-78, is used to pass **Credentials** information between the underlying security mechanisms and the ORB Security Services.

2.5.2.10 The Access Decision Object

The **Access Decision** object is responsible for determining whether the specified credentials allow this operation to be performed on this target object. It uses access control attributes for the target object to determine whether the principal’s privileges, obtained from the **Security Context** are sufficient to meet the access criteria for the requested operation. **Access Decision** objects have the **SecurityLevel2::AccessDecision** Interface as described in Section 2.3.10.2, “The Access Decision Object,” on page 2-104.

2.5.2.11 Audit Objects

There are two types of audit objects:

1. The **Audit Decision** object, which has the **SecurityLevel2::AuditDecision** interface, is used to find out whether an action needs to be audited. Similar audit decision objects are used for all audit policies.
2. The **Audit Channel** objects, which has the **SecurityLevel2::AuditChannel** interface, is used by many of the implementation components (such as interceptors and security objects) and also used by applications to write audit records.

The interfaces are described in Section 2.3.8, “Security Audit,” on page 2-100.

2.5.2.12 Principal Authentication

The **Principal Authenticator** object with the **SecurityLevel2::PrincipalAuthenticator** interface, defined in Section 2.3.3, “Authentication of Principals,” on page 2-73, provides the facility for authenticating a principal. It may also be used by implementation security objects, specifically the **Vault**.

2.5.2.13 Non-repudiation

The Non-repudiation services are accessible through the **NRservice::NRCredentials** interface. Its functionality and operations are defined in Section 2.3.12, “Non-repudiation,” on page 2-108.

2.5.3 Replaceable Security Services

It is possible to replace some security services independently of others.

2.5.3.1 Replacing Authentication and Security Association Services

Replacement of the authentication, security context management, and message protection services underlying a secure ORB implementation can be accomplished by replacing the **Principal Authenticator**, **Vault, Credentials**, and **Security Context** objects with implementations using the new underlying technology.

Note that if the **Vault** uses GSS-API to link to external security services, it may be substantially security technology independent, and so may require no changes or minor changes in order to accommodate a new underlying authentication technology (though it may also have to use technology independent interfaces for principal authentication in some circumstances, as this is not always hidden under GSS-API).

The Vault is replaced by changing the version in the environment.

2.5.3.2 Replacing Access Control Policies

Access control policies can be changed by replacing the **Access Policy** and **Access Decision** objects, which define and enforce access control policies (for example, substituting another **Access Policy** object for **DomainAccessPolicy**).

Applications may also change their access control policies. If the application access policy object(s) is similar to the invocation access policy object(s), then they can be replaced in a similar way.

2.5.3.3 Replacing Audit Services

Audit policies may be replaced, for example, to support certain types of invocation audit policy not supported by the standard audit policy objects. In this case, the policy objects are replaced in a similar way to the access policy objects.

Also, **Audit Channel** objects may be replaced to change how audit records are routed to a collection point or filtered.

The **Audit Channel** object used for object system auditing is replaced by replacing the **Audit Channel** object in the environment. Other **Audit Channel** objects may be replaced by associating a different channel object with the appropriate audit policy.

Application auditing objects can be replaced by the application.

2.5.3.4 Replacing Non-repudiation Services

The Non-repudiation Service is a stand-alone replaceable security service associated with **NRCredentials** and **NRPolicy** objects. Different NR services may use different mechanisms and support different policies. For example, it may be that a service using symmetric encipherment techniques may be replaced by a service using asymmetric encipherment techniques.

The same credentials and authentication method may be used for non-repudiation and for other secure invocations, so when replacing either of these, the effect on the other should be considered.

2.5.3.5 Other Replaceability

No other replaceability points are defined as part of this specification. However, individual implementations may permit replacement of other security services or technologies.

2.5.3.6 Linking to External Security Services

The security service interfaces specified in this section may encapsulate calls to external security services via APIs.

The external services used may include:

- Authentication Services, to authenticate principals.
- Privilege (Attribute) Services, for selecting and certifying privilege attributes for authenticated principals (if access control can be based on privileges as well as on individual identity).
- Security Association Services, for establishing secure associations between applications. These services may themselves use other security services such as Key Distribution Services (if secret keys are used), a Certification Authority for certifying public keys, and Interdomain Services for handling communications between security policy domains.
- Audit (and Event) Services.
- Cryptographic Support Facilities, to perform cryptographic operations (perhaps in an algorithm-independent way).

This specification does not mandate which interfaces are used to access external security services, but notes the following possibilities:

- The GSS-API is used for security associations and for the majority of Credentials and Security Context operations, as this allows easy security service replacement. With this in mind, several interfaces in this specification have been designed to allow easy mapping to GSS-API functions, and the Credentials and Security Context objects are consistent with GSS-API credentials and contexts.
- IDUP GSS-API may be used for independent data unit protection and evidence generation and verification.
- Cryptographic operations performed by a Cryptographic Support Facility (CSF) to ease replacement of cryptographic algorithms. No specific interface is recommended for this yet, as such interfaces are being actively discussed in X/Open and other international bodies, and standards are not yet stable.

Protocols and Mechanisms 3

Contents

This chapter contains the following topics.

3.1 Security Interoperability Protocols

3.1.1 Introduction

This section specifies a model for secure interoperability between ORBs which conform to the CORBA 2 interoperability specification and employ a common security technology.

The interoperability model also describes other interoperability cases, such as the effect on interoperability of crossing security policy domains. However, detailed definitions of these are not given in this specification.

It then defines the extensions required to the interoperability protocol for security. This includes:

- specification of tags in the CORBA 2 Interoperable Object Reference (**IOR**) so this can carry information about the security policy for the target object and the security technology which can be used to communicate securely with it.
- a security interoperability protocol to support the establishment of a security association between client and target object and the protection of CORBA 2 General Inter-ORB Protocol (GIOP) messages between them for integrity and/or confidentiality. This is independent of the security technology used to provide this protection.
- security when using the DCE-CIOP protocol.

As the security information needed by a security mechanism is generally independent of which ORB interoperability protocol is used, other Environment-Specific Protocols (ESIOPs) may support security in a similar way to that described for GIOP. However, the proposal only addresses DCE-CIOP, which supports only DCE security.

The security protocol specified does not define details of the contents of the security tokens exchanged to establish a security association, the integrity seals for message integrity, or the details of encryption used for confidentiality of messages, as these depend on the particular security mechanism used. This specification does not specify mechanisms.

3.1.2 Interoperability Model

This section describes secure interoperability when:

- the ORBs share a common interoperability protocol,
- consistent security policies are in force at the client and target objects, and
- the same security mechanism is used.

All other options build from this.

The model for secure interoperability is shown in the following diagram.

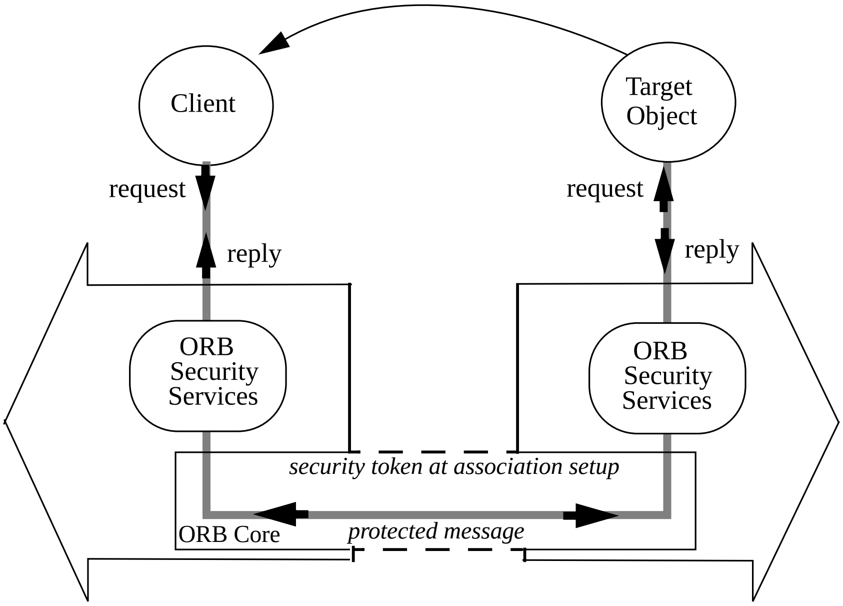


Figure 3-1 Model for Secure Interoperability

When the target object registers its object reference, this contains extra security information to assist clients in communicating securely with it.

The protocol between client and target object on object invocations is as follows:

- If there is not already a security association between the client and target, one is established by transmitting security token(s) between them (transparently to the application).
- Requests and responses between client and target are protected in transit between them. Protection includes not only ensuring individual messages are inviolate and private, but that message streams are as well.

3.1.2.1 Security Information in the Object Reference

When an object is created in a secure object system, the security attributes associated with it depend on the security policies for its domain and object type and the security technology available. A client needs to know some of this information to communicate securely with this object in a way the object will accept. So the object reference transferred between two interoperating systems includes the following information:

- A security name or names for the target so the client can authenticate its identity.
- Any security policy attributes of the target relevant to a client wishing to invoke it. This covers policies such as the required quality of protection for messages and whether the target requires authentication of the clients identity and supports authentication of its identity.

- Identification of the security technology used for secure communication between objects this target supports and any associated attributes. This allow the client to use the right security mechanism and cryptographic algorithms to communicate with the target.

3.1.2.2 Establishing a Security Association

The contents of the security tokens exchanged depend on the security mechanism used.

A particular security mechanism may itself have options on how many security tokens are used. The minimum is an *initial context* token (a term used in GSS-API), sent from the client to the target object to establish the security association. This typically contains:

- an identification of the security mechanism used,
- security information used by this mechanism to establish the required trust between client and target and to set up the security context necessary for protecting messages later,
- the principal’s credentials, and
- information for protecting this security data in transit.

In addition to this token, subsequent security tokens may be needed if:

- mutual authentication of client and target object is required, or
- some negotiation of security options for this mechanism is required (for example, the choice of cryptographic algorithms).

3.1.2.3 Protecting Messages

The invocation may be protected for integrity and/or confidentiality. In either case, the messages forming the request and reply are first wrapped in a sequencing layer envelope and then cryptographically protected by the ORB security services. For integrity, extra information (e.g., an integrity seal) is added to the message so the target ORB security services can check that the message has not been changed.

For confidentiality, the message itself is encrypted so it cannot be intercepted and read in transit.

Details of how messages are protected are again mechanism-dependent. Note, however, that messages cannot be changed once they have been protected, as they cannot be understood once confidentiality protected and the integrity check will fail if they are altered in any way.

In SECIOP message stream protection is provided by encapsulating all SECIOP data payloads (e.g., IIOP messages or message fragments) in a sequencing protocol frame. The sequencing protocol ensures that data payloads are not duplicated (replayed), dropped (deleted), or received out-of-sequence (reordered). The sequencing protocol frame is protected by the ORB security services to ensure the state it contains is not modified by an intruder.

3.1.2.4 Security Mechanisms for Secure Object Invocations

The interoperability model above can be supported using different security mechanisms.

This specification does not define a standard security mechanism to be supported by all secure ORBs. It therefore does not specify a particular set of security token formats and message protection details for a particular security mechanism.

3.1.2.5 Security Mechanism Types

There are two major types of security mechanisms used in existing systems for security associations. They are those using:

- Symmetric (secret) key technology where a shared key is used by both sides, and a trusted third party (a Key Distribution Service) is used by the client to obtain a key to talk to the target.
- Asymmetric (public) key technology where the keys used by the two sides are different, though linked. In this case, long term, public keys are normally freely available in certificates which have been certified by a Certification Authority.

Several existing systems use symmetric key technology for key distribution when establishing security associations. These are usually based on MIT's Kerberos product. Such systems normally include no public key technology.

Other security mechanisms use public key technology for authentication and key distribution as this has advantages for scalability and inter-enterprise working. The number of public key based systems are growing and the use of public key technology is standard for non-repudiation, which is an optional component in this specification, and increasingly needed in commercial systems so any OMG security specification must not preclude its use. Also, the use of smart cards with public key technology is increasing. However, non-repudiation is not a service required for secure interoperability.

Interoperating with Multiple Security Mechanisms

The current specification allows a client to identify the security mechanism(s) supported by the target. Where a client or target supports more than one mechanism, and there is at least one mechanism in common between client and target, the client can choose one which they both support.

Some security mechanisms may support a number of options, for example:

- a choice of cryptographic algorithms for protecting messages,
- a choice of using public or secret key technology for key distribution.

The appropriate options can be chosen by the client in the same way as choosing the basic mechanism, via the client security policy and information in the target's object reference. However, some mechanisms will be able to negotiate options using extra exchanges at association establishment which are specific to the particular mechanisms.

Interoperating between Underlying Security Services

Security mechanisms for secure object invocations use underlying security services for authentication, privilege acquisition, key distribution, certificate management, and audit. Under some circumstances, these need to inter-operate. For example, key distribution services may need to communicate with each other, and audit services may need to transmit audit records between systems.

Interoperability of such underlying security services is considered out of scope of this specification, as they are mechanism dependent.

3.1.2.6 Interoperating between Security Policy Domains

The sections above consider interoperability within a security policy domain where consistent security policies apply to access control, audit and other aspects of the system. These rely on information about the principal, including its identity and privilege attributes, being trusted and having a consistent meaning throughout the policy domain.

Where a large distributed system is split into a number of security policy domains, interoperation between security policy domains is needed. This requires the establishment of trust between these domains. For example, an ORB security association service at a target system will need to identify the source of the principal's credentials so it can decide how much to trust them.

Once the identity of the client domain has been established, interdomain security policies need to be enforced. For example, access control policies are mainly based on the principal's certified identity and privilege attributes. The policy for this could be:

1. The target domain trusts the client domain to identify principals correctly, but does not trust their privilege attributes, so treats all principals from other domains as guest users.
2. The administrators of the two domains have agreed some privilege attributes in common, and trust each other to give these only to suitably authorized users. In this case, the target system will give principals from the client domain with these privileges the same rights as principals from the target domain.
3. The administrators of the two domains agree what particular privilege attributes in the client domain are equivalent to particular privilege attributes in the target domain, and so grant corresponding access rights.

For the first two of these, the target domain security policy could enforce restrictions about which privilege attributes may be used there. This would not necessarily affect the interoperability protocols - the **get_attributes** operation will simply not return all of the privileges. But even in this case, some security mechanisms will choose to modify the principal's credentials to exclude unwanted attributes.

In the third case, the privilege attributes need to be translated to the ones used in the target domain. If this translation is to be done only once, an interdomain service is likely to be used which both translates the credentials and reprotects them so they can be delegated between nodes in the target domain.

Such an interdomain service may be invoked by the ORB Security Services, but may be invoked by a separate interoperability bridge between the ORB domains. If invoked by an ORB service, it extends the implementation of the **Vault** object described previously and this will probably call on a mechanism specific Interdomain Service.

3.1.2.7 Secure Interoperability Bridges

Secure Interoperability Bridges between ORB domains are relevant to this architecture, as in future, they may be specified as part of some secure CORBA compliant systems. However, this section does not describe how to build such bridges.

Secure interoperability bridges may be needed for:

- ORB-mediated bridges, where data marshalling is done outside the ORB and associated ORB services.
- Translating between security mechanisms (technology domains).
- Mapping between security policy domains.

In all these cases, both the system and application data being passed will need to be altered, affecting its protected status. This needs to be re-established using security services trusted by both client and target domains.

3.1.3 Protocol Enhancements

The following sections detail the enhancements required to the CORBA 2 interoperability specification for security.

Section 3.1.4, “CORBA Interoperable Object Reference with Security,” on page 3-7 defines the enhancements needed to the Interoperable Object Reference (IOR).

Section 3.2, “Secure Inter-ORB Protocol (SECIOP),” on page 3-34 defines the enhancements needed to secure GIOP messages and Section 3.8, “DCE-CIOP with Security,” on page 3-105 defines the DCE-CIOP with security.

3.1.4 CORBA Interoperable Object Reference with Security

The CORBA 2 Interoperable Object Reference (**IOR**) comprises a sequence of ‘tagged profiles.’ A profile identifies the characteristics of the object necessary for a client to invoke an operation on it correctly, including naming/addressing information. The tag is a standard, OMG-allocated identifier for the profile which allows the client to interpret the profile data, but although the tag is OMG-allocated, the profile itself may not be OMG-specified.

A multi-component profile is a profile that itself consists of tagged components. This specification defines TAGS for use in such multi-component profiles as follows:

The following TAGs are defined:

- **IIOP components**, which can be used in a multi component profile (see Appendix B, Section B.8, “Secure Inter-ORB Protocol (SECIOP),” on page B-21.

- **Security components** that identify security mechanism types, one for each mechanism supported. Each security mechanism component can also include mechanism specific data.
- Aspects of the target object policy that cover the dependencies between and overall use of components (for example, the quality of protection required) may be specified in separate **policy components**. This avoids establishing unnecessary dependencies between other (technology) components.

Use of tagged components within the multi component profile to carry IIOP, security and other data may cause performance degradations in certain situations. For example, if an **IOR** carries many tagged components that are unrecognized by a client implementation, it must process these when they appear before those that it does recognize. Some, such as the components describing IIOP, have a high probability of being recognized and used by many clients. Consequently, implementations with an objective to optimize IOR processing will place such components at the beginning of the tagged component sequence.

3.1.4.1 Security Components of the IOR

The following new tags are used to define the security information required by the client to establish a security association with the target. Note that a tag may occur more than once, denoting that the target allows the client some choice. All tag component data must be encapsulated using CDR encoding

TAG_x_SEC_MECH

This is the prototype TAG definition for OMG registered security association mechanisms. The mechanism is identified by the TAG value. The component data for TAGs of this kind is defined by the person who registers the TAG. The confidentiality and integrity algorithms to be used with the mechanism may either be encoded into the TAG value or in mechanism specific data (see Appendix G, Section G.2, “Guidelines for Mechanism TAG Definition in IORs,” on page G-1).

If this definition includes

sequence <TaggedComponent> components;

then the components field can contain any of the other component TAGs, whose values can be specific to the mechanism.

If the mechanism is selected for use, the components in this field are used in preference to any recorded at the multi component level.

Multiple **TAG_x_SEC_MECH** components may be present to enumerate the security mechanisms available at the target.

TAG_GENERIC_SEC_MECH

This TAG enables mechanisms not registered with the OMG, but common to both client and target to be used with the standard interoperability protocol. Its definition is:

```
struct GenericMechanismInfo {
    sequence <octet> security_mechanism_type;
    sequence <octet> mech_specific_data;
    sequence <TaggedComponent> components;
};
```

The first part of this TAG is the **security_mechanism_type** which identifies the type of underlying security mechanism supported by the target including confidentiality and integrity algorithm definition. It is an ASN.1 Object Identifier (OID) as described for use with the GSS-API in IETF RFC 1508.

The **mech_specific_data** field allows mechanism specific information to be passed by the target to the client.

The components field can contain any of the other component TAGs, whose values can be specific to the mechanism.

If the mechanism is selected for use, the components in this field are used in preference to any recorded at the multi component level.

Multiple **TAG_GENERIC_SEC_MECH** components may be present to enumerate the security mechanisms available at the target.

TAG_ASSOCIATION_OPTIONS

This TAG is used to define the association properties supported and required by the target. Its definition is:

```
struct TargetAssociationOptions{
    AssociationOptions    target_supports;
    AssociationOptions    target_requires;
};
```

target_supports - gives the functionality supported by the target

target_requires - defines the minimum that the client must use when invoking the target, although it may use additional functionality supported by the target.

The following table gives the definition of the options.

Table 3-1 Definition of Association Options

Association Options	target_supports	target_requires
NoProtection	unprotected messages	the minimal protection requirement is unprotected invocations.
Integrity	integrity protected messages	messages to be integrity protected
Confidentiality	confidentiality protected invocation	invocations to be protected for confidentiality

Table 3-1 Definition of Association Options *(Continued)*

Association Options	target_supports	target_requires
DetectReplay	the target can detect replay of requests (and request fragments)	security associations to detect message replay
DetectMisordering	target can detect sequence errors of requests and request fragments	security associations to detect message mis-sequencing
EstablishTrustInTarget	the target is prepared to authenticate its identity to the client	(this option is not defined)
EstablishTrustInClient	the target is capable of authenticating the client	establishment of trust in the client’s identity
NoDelegation	target supports no delegation	the target states that delegation will not be supported
SimpleDelegation	the target supports simple delegation	(this option is not defined)
CompositeDelegation	the target supports composite delegation	(this option is not defined)

TAG_SEC_NAME

The target security name component contains the security name used to identify and authenticate the target. It is an octet sequence, the content and syntax of which is defined by the authentication service in use at the target. The security name is often the name of the environment domain rather than the particular target object.

The **TAG_SEC_NAME** component is not needed if the target does not need to be authenticated.

3.1.4.2 IOR Example

Table 3-2 IOR Example

tag	value	mech specific tag	value
tag_sec_name	"Manchester branch"		
tag_association_options	supports and requires integrity and to establish trust in the clients privileges		
tag_generic_sec_mech	mech 1 oid		
		tag_sec_name	"MBn1"
		tag_association_options	supports and requires integrity, replay detection, misordering detection, and to establish trust in the client's security attributes
tag_generic_sec_mech	mech 2 oid		
		tag_association_options	target requires and supports confidentiality and to establish trust in the client's security attributes

In this example, if mechanism "mech 1" is used, the target security name is "MBn1" while the association must use integrity replay and misordering options. If mechanism "mech 2" is used, no mechanism specific security name has been specified and so "Manchester branch" is used as the security name. The association options are EstablishTrustInClient and Integrity.

3.1.4.3 Operational Semantics

This section describes how an ORB and associated ORB services should use the IOR security components to provide security for invocations and how the target object information should be provided.

Client Side

During a request invocation, the non-security tagged components in the **IOR** multi-component profile indicate whether the target supports IIOP and/or some other environment specific protocol such as DCE-CIOP. Security mechanism tag components specify the security mechanisms (and associated integrity and confidentiality algorithms) this target can use. The ORB selects a combination of interoperability protocol and security mechanism that it can support.

If there is a common interoperability protocol, but no common security mechanism, then a secure request on this **IOR** cannot be assured.

If the same security mechanism is supported at the client and the target, but the **TAG_ASSOCIATION_OPTIONS** component specifies no protection is needed or no **SEC_MECH** is specified, then unprotected requests are supported by the target, and the request can be made without using security services. If the target requires protected requests, then the ORB must choose an alternative transport and/or security mechanism.

The **IOR** tags and the client's policies and preferences are used together to choose the security for this client's conversation with the target.

The specific security service used may not understand the CORBA security values, and so may require them to be mapped into values it can understand.

Determining Association Options

The **Association Options** in Table 3-1 on page 3-9, lists possible association options such as **NoProtection**, **Integrity**, **DetectReplay**.

The actual association options used when a client invokes a target object via an **IOR** depend on:

- The client-side secure invocation policy and environment.
- Client preferences as specified by **set_association_options** on the **Credentials** or **set_policy_overrides** of the object reference invoked with a **QOPPolicy** object as one of the Policies to be overridden.
- The target-side secure invocation policy and environment (as indicated by information in the **TAG_ASSOCIATION_OPTIONS** component).

An association option should be enforced by the security services if the client requires it and the target supports it, or the target requires it and the client supports it.

If the target cannot support the client's requirements, then a CORBA::NO_PERMISSION exception should be raised. If the client cannot meet the requirements of the target, then the invocation may optionally proceed, allowing policy enforcement on the target side.

Target Side

The security information required in the **IOR** for this target must be supplied from the target (or its environment). This specification does not define exactly when particular information is added, as some of it may only be needed when the object reference is exported from its own environment.

The security information may come from a combination of:

- The object's *own credentials* (see Section 2.3.7, "Security Operations on Current," on page 2-93). This includes for example, the target's security name. It could include mechanism specific information such as the target's public key if it has one.
- Policy associated with the object. This includes, for example, the QOP.
- The environment. This includes, for example, the mechanism types supported.

The target object does not need to supply this information itself. This is done automatically by the ORB when required. For example, much of the information for the target's own credentials are set up on object creation.

As at the client, the specific security service used may require CORBA security values to be mapped into those it understands.

If when the client invokes the target identified by the **IOR** an **Invoke Response** message is returned for the request with the status **INVOKE_LOCATION_FORWARD**, then the returned multiple component profile must contain security information as well as the new binding information for the target specified in the original **Invoke Request** message.

Any security information in the returned profile applies to the new binding information and replaces all security information in the original profile. This **INVOKE_LOCATION_FORWARD** behavior can be used to inform the client of updated security information (even if the address information hasn't changed).

3.1.5 Common Secure Interoperability Levels

Three Common Secure Interoperability Levels are defined to help in classifying and positioning the various interoperability facilities that are defined, and also to help in concisely stating the conformance requirements. The three CSI levels are:

CSI Level 0 - supports only identity based policies without delegation.

CSI Level 1 - supports identity based policies with or without unrestricted delegation.

CSI Level 2 - supports identity and privilege based policies with controlled delegation.

A complete description of the these CSI levels of interoperability can be found in Appendix D, Section D.7.2, "Common Secure Interoperability Levels," on page D-12.

3.1.6 Key Distribution Types

Security mechanisms use cryptography in the establishment of a secure association between a client and target and in protecting the data between them. Security mechanisms differ in the type of cryptography they use, particularly for distribution of keys. (Keys are assigned to clients, targets, and trusted authorities). Three types of key distribution are defined in this specification:

- **Secret keys** - use secret key technology for distribution of keys for principals.

- **Public keys** - use public key technology for distribution of keys for principals, though may use secret key technology for message protection.
- **Hybrid** - use secret key technology for key distribution for principals within an administration domain, and public key technology for key distribution for trusted authorities, and hence between domains.

All types of key distribution can be used to support all the facilities in CORBA Security for secure object invocations (though public key is almost universally used for non-repudiation). The choice of mechanism to use depends on a customer's requirements. For example, to fit with other systems and for scalability to inter-enterprise working.

3.1.7 Security Mechanisms Hosted on SECIOP

The choice of protocol to use depends on the mechanism type required and the facilities required by the range of applications expected to use it. How the mechanisms underlying the following three security protocols are hosted on SECIOP are specified:

3.1.7.1 1. SPKM Protocol

Supports identity based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on the definition in [20].

3.1.7.2 2. GSS Kerberos Protocol

Supports identity based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0).

The GSS Kerberos protocol is based on the [12] which itself is a profile of [13].

3.1.7.3 3. CSI-ECMA protocol

Supports identity and privilege based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0).

For keys assigned to principals, it has two options:

- It can use either secret or public key technology.
- It uses public key technology for keys assigned to trusted authorities.

The CSI-ECMA protocol is based on the ECMA GSS-API Mechanism as defined in ECMA 235, but is a significant subset of this - the SESAME profile as defined in [16]. It is designed to allow the addition of new mechanism options in the future; some of these are already defined in ECMA 235.

The following table shows which CSI functionality is supported with which protocols.

Table 3-3 CSI Functionality and Protocols

Protocol CSI Level	SPKM	GSSKerberos	CSI-ECMA
0	Supported	Supported	Supported
1	Not supported	Supported (Mandatory)	Supported
2	Not supported	Not supported	Supported

3.1.8 Security Mechanisms Hosted Directly on IIOP

The SSL [21] protocol which provides for confidentiality and integrity within the IP sockets paradigm can be used to provide interoperability based on this protocol hosted directly on IIOP. How this is done is specified in Section 3.7, "Integrating SSL with CORBA Security," on page 3-103. It supports identity based policies without delegation.

3.1.9 Choices of Protocols, Cryptographic Profiles and Key Technologies

What combination of Security Protocols, Key Technologies, and Cryptographic Profiles are the most desirable has been subject of debate both inside and outside OMG. In this specification, certain choices have been made based on the belief that these choices best meet OMG's current needs given the other constraints.

3.1.9.1 Choice of Protocol and Key Technology

GSS Kerberos is specified as the mandatory protocol for common secure interoperability, as Kerberos is widely available and most vendors can support it. However, it does not provide all facilities required and is secret key only.

Several other protocols are specified as non-mandatory options follows:

- CSI-ECMA is specified as a protocol to provide support for the full set of CORBA security facilities using public key or secret key technology.
- SPKM is specified as a simpler public key protocol suitable for applications where
  - access and audit policies are static, and
  - at each stage in a chain of object invocations, the policies depend only on the identity of the immediate invoker, not the initiator of the chain.
- SSL is specified for use in the web market.

3.1.9.2 Cryptographic Profiles

Security mechanisms use cryptography in the establishment of a secure association between a client and target and in protecting the data between them. Different cryptographic algorithms are used to support particular security functions depending on the type of mechanism used and also the regulations on use of cryptography. The combination of algorithms used to provide particular security using a particular mechanism is called a cryptographic profile.

Currently, different cryptographic algorithms, and/or different key lengths are required to meet export controls and regulations on use of cryptography in various countries (see "International Deployment" on page 3-17). Although some vendors produce more than one version of secure products for different markets, they are increasingly reluctant to do this. For common secure interoperability, a particular cryptographic profile is needed. Some options are to standardize:

- Integrity only for user data, not confidentiality. If done using **MD5**, this is likely to be exportable and generally deployable, but doesn't provide confidentiality when interoperating. This does not provide the functionality which some users will want.
- Integrity and confidentiality using weak keys only. This provides the required functionality, in a way which can generally be exported, but does not provide the strength of protection needed by some customers. Also, products using it may be subject to import controls or other regulations in some countries.
- On strong confidentiality and integrity, which customers want, but will be subject to export controls in most countries and to deployment regulations in some. Leave vendors and customers to sort out the problems.

This chapter makes only the first of these options mandatory; however, implementors of all profiles may choose to support other profiles also.

3.1.9.3 Conformance to External Security Mechanisms

This specification uses protocols defined in other standards documents. It refers to particular versions of these standards, which is needed for interoperability. If the versions of these external documents change in future, there may be a need to update this specification so that it is in line with the most accepted external version of these standards.

3.1.10 Common Secure Interoperability Requirements

This section describes the requirements that Common Secure Interoperability is expected to meet.

The Common Secure Interoperability specification is required to provide for standard security mechanisms, simple delegation, and international deployment. This section discusses the key requirements for common secure interoperability that have driven the design of this specification and how this specification responds to these requirements.

3.1.10.1 CORBA Standard Security Mechanisms

Standard CORBA security mechanisms are required so that ORBs can interoperate securely at all.

Four popular security mechanisms to meet different circumstances, as described above, can be used to host CORBAsecurity in a standard way. One of the four described in this chapter is mandatory and all conformant ORBs must support it. Interoperability between conformant ORBs is always possible using this; however, the facilities supported when using it are limited.

Interoperability also requires common use of cryptographic algorithms. A number of cryptographic profiles are specified to meet the needs of different markets and countries. One is mandatory and interoperability between conformant ORBs is always possible using this; however, it provides data integrity but not confidentiality.

Where multiple mechanisms and cryptographic profiles are supported by both ORBs, the client and target object must agree which to use. In this specification, this is done by the client looking at the security mechanism tag in the target object reference and choosing an appropriate mechanism and profile which both support. (In future, negotiation of mechanisms may be supported.)

3.1.10.2 International Deployment

International deployment requires that the security mechanisms and algorithms chosen can be used worldwide in countries which are subject to different national regulatory controls on the use of cryptography. It also requires that they can be used across international boundaries. International deployment may also be affected by export control regulations and other issues.

Requirements distilled from the key regulations affecting international deployment include:

- Keeping the amount of information which must be encrypted for confidentiality to a minimum. In general, encryption of keys is acceptable, but encryption of other data may not be. For this reason, encryption of security attributes is undesirable. At CSI level 2, where more attributes are generally needed, the part of the security tokens concerned with key distribution is separated from the part used to carry privileges (e.g., in CSI-ECMA); therefore, the latter part does not have to be encrypted.
- Being able to use identities for auditing which are anonymous, except to the auditor. For this reason, identities used for access control and audit may need to be different. A separate **AuditId** can be transmitted at level 2.
- Allowing use of different cryptographic algorithms, with different lengths of keys for specified functions to meet export and use regulations in different countries. The specification defines cryptographic profiles which allow for different cases. The mandatory one provides data integrity only, as this is generally easier to deploy internationally.

There may be further requirements on secure ORB products to ensure that they are exportable. For example, they must not allow easy/uncontrolled replacement of cryptographic algorithms. This affects the construction of the system, but not this interoperability standard, so is not considered further here.

Other restrictions on the use of algorithms and security mechanisms are highlighted in "Identifying Encumbered Technology" on page 3-20. For example, the DES algorithm is subject to export controls, while RSA requires licensing in some countries. The MIT version of the Kerberos technology, widely used in the USA, is also subject to export controls.

3.1.10.3 Consistency

It should be possible to provide consistent security across the distributed object system and with associated legacy and other non-object systems. This includes:

- Support of consistent policies for which principals should be able to access the sort of information, within a security domain, that includes heterogeneous systems.

  For this specification, it requires the ability to transmit consistent privilege and other attributes between ORBs to support these policies. Level 0 and 1 conformant ORBs can transmit identities, level 2 conformant ORBs can transmit a range of privilege attributes. These can be the ones used in existing systems, though system specific ones will not be usable in other systems.

- Fit with existing logons (so extra logons are not needed) and with existing user databases (to reduce the user administration burden).

  Log on needs to result in credentials which include the information required to support the specified security mechanisms. Note that single logon with secure messaging, web, etc. generally requires use of public key based mechanisms. Also, if non-repudiation is supported, they will also need to include the security information required to support the non-repudiation mechanism (normally, a public key mechanism).

  Also, interoperating with non-object systems may require, for example, a CORBA object implementation which calls a non-CORBA application to be able to delegate incoming credentials (assuming compatible security mechanisms.)

- Fit with all non-object systems is clearly not possible if such a system uses security mechanisms which are incompatible with the one used in the object system. Such systems may be able to use CORBA Security, but will not be able to interoperate using the common secure interoperability standard.

This specification includes an interoperability level which supports privileges and a public key (as well as a secret key) mechanism to support these requirements.

3.1.10.4 Scalability

It should be possible to provide security for a range of systems from small, local systems to large intra- and inter-enterprise systems. For larger systems, it should be possible to:

- Base access controls on the privilege attributes of users such as roles or groups (rather than individual identities) to reduce administrative costs. This specification includes the transmission of such privilege attributes in CSI level 2.

- Have a number of security domains which enforce different security policy details, but support interworking between them subject to policy. (This specification includes the architecture for such inter-domain working, though this specification does not define interface for this.) Use of public key technology helps large scale, particularly inter-enterprise interoperability.
- Manage the distribution of cryptographic keys across large networks securely and without undue administrative overheads.

3.1.10.5 Flexibility of Security Policy

The security policies required varies from enterprise to enterprise, so choices should be allowed, though standard policies should be supported for common secure interoperability.

Access Policies

At CSI levels 0 and 1, the **AccessId** is the only privilege attribute supported. The standard **DomainAccessPolicy** defined in Section 2.4.4, “Access Policies,” on page 2-119 (or other access policies) can be used with only this privilege.

At CSI level 2, conformant ORBs are able to transmit further privilege attributes (such as role and group), so the **DomainAccessPolicy** (and other access policies) can be used with these privileges also.

CSI level 2 is designed to allow transmission of further privileges, including user defined privileges and security clearances as needed for multi-level secure systems. If received by a conformant ORB, they will be available for access control at the target. However, conformant ORBs need not transmit them, so use of such privileges is subject to the agreement between the systems.

The mechanisms defined here also allow a wider range of privileges, etc. to be supported and other access policies to be used. However, interoperability with all other conformant ORBs is not guaranteed in this case.

Audit Policies

All CSI levels provide an **AuditId** that can be used in audit policies. CSI level 2 can transmit an **AuditId**, which is anonymous to all but audit administrators.

3.1.10.6 Application Portability

Application portability is an important OMG requirement. The many applications which are unaware of security will continue to be portable.

Applications which enforce their own security policies should still be portable across ORBs supporting common secure interoperability if the access and audit policies they use rely only on security attributes which are mandatory in the chosen CSI level.

Applications should be unaware of the security mechanism used to enforce the security, unless they specifically ask what it is (e.g., using **get_service_information**, see Section 2.3.2, “Finding Security Features,” on page 2-73).

3.1.10.7 Security Services Portability/Replaceability

The CORBA Security specification includes replaceability conformance options.

The objects supporting the security mechanism (**PrincipalAuthenticator**, **Vault**, and **Security Context**) can be replaced to support the mechanisms in this specification. However, if logon outside the object system is supported, this will need to provide credentials including the security information needed by the CSI mechanism(s) used.

If the invocation access policy is replaced, this can utilize privileges transmitted using CSI protocols. However, if an ORB wishes to control access on invocations using local (e.g., operating system) attributes, then mapping of attributes prior to calling the **Access Decision** object is needed.

3.1.10.8 Performance

Security should not impose an unacceptable performance overhead, particularly for normal commercial levels of security, although a greater performance overhead may occur as higher levels of security are implemented.

Details of the performance overhead depend on the mechanism used and its implementation; however, in this specification:

- Sufficient information can be carried in the **IOR** so that the client knows what security the target supports and does not have to negotiate protocols and options with it.
- The mechanisms used allow the **initial_context_token** to be transmitted with first message, if mutual authentication is not required.

3.1.10.9 Identifying Encumbered Technology

This specification includes technology which is encumbered to some extent.

- The Kerberos V5 technology is licensable from the Massachusetts Institute of Technology without cost and is widely deployed within the USA. However, it is subject to export control from the USA; therefore, [12] is the definition of the protocol used here, as this can be implemented independently of the MIT Kerberos code.
- SPKM implementations are available, though not free. As for other mechanisms, the (draft) standard is the basis of this specification.
- SESAME implementation is available, but is not free for commercial use, and has restrictions on cryptography for export reasons (the public version does not include commercial cryptographic profiles - it has the secret key algorithm replaced by XOR for export control reasons).

- There are two patents associated with the CSI-ECMA protocol. These are usable free of charge for implementations conformant with this specification under fair conditions (formal definition of these are available from Bull and ICL).
- The DES algorithm is widely deployed internationally, but is subject to export controls. Export with key lengths which provide strong confidentiality is not generally permitted.
- Increasingly, the RSA algorithm is widely deployed internationally; however, it is subject to licensing in the USA. It is also subject to export controls, though where it can be shown that it is not used for confidentiality, products using it are more likely to be exportable.
- Any other cryptographic algorithms used are generally subject to export controls, as is any interface which makes it easy to replace algorithms.

3.1.11 Relation to CORBA Security Facilities and Interfaces

This section describes how the security facilities and interfaces defined in Sections 2.3 through 2.5 map to various elements of security protocol mechanisms. It is aimed at:

- Object implementors developing applications using a secure object system who need to know what security is available.
- Implementors of security policies who may be constrained by the security attributes available when interoperating according to this standard.
- ORB implementors supporting replaceable security policies.

3.1.11.1 Functionality

The security information that is transmitted between ORBs, and which security facilities and policies are supported in an interoperable environment, is described in these sections. Three levels of secure interoperability are defined specifying the particular security attributes that conformant ORBs must support.

Note that the interoperability defined here is for interoperability of requests/responses between ORBs. It does not include interoperability of the evidence tokens used for non-repudiation.

3.1.11.2 Replaceability

In replaceability, options which allow ORB implementors to support a wide range of security policies and mechanisms is defined. For example, the standard **DomainAccessPolices** can be replaced by other policies where ORBs support the appropriate replaceability option. This specification still allows this replaceability, though the policy being added may be restricted by the security information guaranteed to be available.

This specification allows replaceability of security mechanisms by replacement of the **Vault** and **Security Context** objects. It specifies mechanisms and protocols which can be implemented via a GSS-API interface. This adds the potential for having a single implementation of the **Vault** and **Security Context** objects, which by using GSS-API, would be able to use different security mechanisms.

3.1.11.3 Levels of Interoperability

This specification includes three interoperability levels, as described more completely in Appendix D, Section D.7.2, "Common Secure Interoperability Levels," on page D-12. This section gives information about these levels and an example showing the difference in the way they handle a particular problem.

Common Secure Interoperability Level 0

CSI level 0 supports identity based policies without delegation. It requires ORBs to support the following:

- Authentication of principals using security functions under one ORB, and then use of the resultant credentials when making a secure invocation to an object under a different ORB.
- Secure associations to establish trust between client, target, and protect messages.
- As part of the secure association, the security name of the client is passed to the target and used to set both **AccessId** and **AuditId** so that identity based access and audit policies can be supported.

  The identity is always that of the immediate invoker of an object in a chain of object invocations, this is only the same as the initiator of the chain at the point of entry to the chain.

Common Secure Interoperability Level 1

CSI level 1 supports identity based policies with unrestricted delegation. It requires ORBs to support the mandatory part of the CORBA Security when two conformant ORBs interoperate (using the same security mechanism). It provides the CSI level 0 facilities plus security information (in particular, the security name) of a principal in the call chain can be delegated to objects (subject to security policy).

Once this security information has been delegated, the intermediate object has the choice of acting under its own identity or delegating the initiating principal's identity when invoking another object. When delegating another principal's identity, the delegated identity (rather than the immediate invoker's identity) is used to set both the **AccessId** and **AuditId** at the target.

Common Secure Interoperability Level 2

CSI level 2 supports identity and privilege based policies with controlled delegation. ORBs supporting this level must support interoperability of all facilities in Sections 2.3 through 2.5 concerned with object invocation. CSI level 2 provides the CSI level 0 and level 1 facilities plus:

- The security information of the immediate invoker or the delegated information of the initiating principal can include more security attributes, as follows:
  - an extensible range of privilege attributes (e.g., roles, groups, enterprise defined attributes) to support a wider range of policies. Generally, these attributes include an **AccessId** which is independent of the security name (and the mechanism type used) and is used to set the **AccessId** at the target. Interoperability using particular types of privileges depends on these privileges being common to both ORBs. This CSI specification defines which privileges a CSI level 2 conformant ORB must support (see Appendix D, Section D.7.2, “Common Secure Interoperability Levels,” on page D-12.
  - a separate **AuditId** can be transmitted. This may be anonymous (except to the audit administrator). It will always represent the actual principal using the system, even when the **AccessId** represents someone who has allowed another user to access the system on his behalf.
- The delegation of a principal’s attributes can be controlled (for example, usable at only identified (groups of) targets). Intermediate receiving delegated security attributes of a principal will not always be able to delegate them.
- Composite delegation is allowed for, but support for this is not mandatory.

Example

This section looks at an example of a secure object system which highlights the difference between the delegation facilities of the three CSI levels. In this example, Bob wants to close his bank account and is prepared to give Dan power of attorney to do this.

- At CSI level 0, no delegation is possible; therefore, Bob has to go to the bank and close the account himself.
- At CSI level 1, Bob gives Dan unlimited power of attorney to act for him (as delegation is unrestricted). Dan can close Bob’s bank account. As the power of attorney is unlimited, Dan can also read Bob’s medical records and pass on the power of attorney to Mark - who can also close Bob’s bank account, read Bob’s medical records, etc.
- At CSI level 2, Bob gives Dan the power of attorney to close his bank account; therefore, Dan can close the account. But this does not include the right to read Bob’s medical records (as only limited privileges were given to Dan) and does not include the right to give the power of attorney to Mark (as delegation was restricted to Dan).

3.1.12 Security Functionality

This section reviews the security functionality in Section 2.3, “Application Developer’s Interfaces” through Section 2.5, “Implementor’s Security Interfaces” and specifies which functionality is supported interoperably at which CSI level. Some security functionality is supported at all CSI levels, some only at CSI level 1 or 2.

3.1.12.1 Authentication

The CSI mechanisms do not specify authentication of principals, but use the result of such authentication. Principal authentication must result in credentials which contain the security information needed by the security mechanisms supported by this conformant ORB.

CSI mechanisms require authenticated principals (see Section 2.3.3, “Authentication of Principals,” on page 2-73).

3.1.12.2 Access Control

Access controls depend upon the privileges of the principal.

At CSI levels 0 and 1, only the principal’s identity is available at the target; therefore, Access Policies using this level must either:

- use only the principal’s identity for access control, or
- retrieve other attributes for that principal prior to taking the access decision (the “pull” model).

The standard **DomainAccessPolicy** assumes all privileges required have been “pushed” from the client; therefore, they will be restricted to using identity only. Access policies using the pull model will not be portable, if the source of such attributes is system dependent.

At CSI level 2, the **AccessPolicies** can use any of the privileges supported by both ORBs. All CSI level 2 conformant ORBs support **AccessId**, **GroupId**, and **Role**. They may also transmit user defined privileges, where the user enterprise concerned has a CORBA attribute family definer, and defines its own families of attributes. However, some attribute types defined outside the object system may not be understood at all targets; therefore, portability of these may not be possible to all environments.

3.1.12.3 Audit

Auditing is as defined in Section 2.1.5, “Auditing,” on page 2-11, and is possible at all CSI levels. A separate **AuditId** (which may be anonymous) can be transmitted at CSI level 2.

3.1.12.4 Secure Invocation

Conformant implementations (all CSI levels) must support all the association options defined in Table 3-1 on page 3-9.

Channel bindings, as defined in GSS-API and all protocols defined here, are not part of the mandatory specification.

Conformant implementations at level 2 allow use of algorithms with different strengths for integrity and confidentiality.

3.1.12.5 Delegation Facilities

- At CSI level 0, no delegation is supported.
- At CSI level 1, the initiating principal’s identity can be delegated to the target. It is either delegated or not - there are no other restrictions on delegation.
- At CSI level 2, the initiating principal’s privileges, as well as identity, can be delegated to the target. Delegation can be controlled further, restricting the targets to which the attributes can be delegated. These restrictions must be specified by administrative action, as there are no interfaces specified in to do this in this specification.

Level 2 protocols are also defined which allow support of composite delegation; however, support of this is not required by conformant ORBs.

3.1.12.6 Non repudiation

Non-repudiation relies on NR credentials for handling NR evidence tokens. The same credentials can be used for secure invocations and non-repudiation. This will only be possible if compatible security technology is used for non-repudiation and secure invocation. While no specific security technology is mandated for non-repudiation, it is expected that this will use public key technology. Common credentials usable for both purposes are expected to use public key technology, to fit with public key mechanisms (SPKM or the CSI-ECMA public key option), rather than with secret key mechanisms.

3.1.12.7 Security Policies

Security policies are potentially sharable between ORBs if they use only identities and privileges which are available at both ORBs and can be transmitted between them. For example, a **DomainAccessPolicy** that uses roles must receive requests from an ORB which can generate them via a CSI level 2 protocol which can transmit roles.

3.1.13 Model for Use and Contents of Credentials

The CORBA Security model includes security functionality enforced during object invocations and by applications, as shown in Figure 3-2.

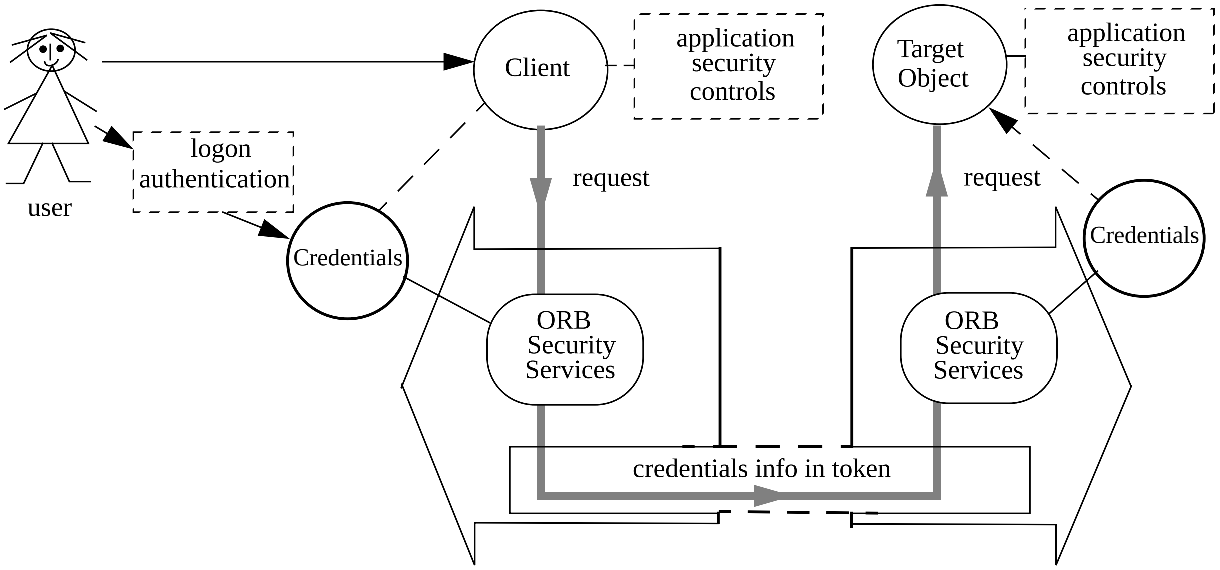


Figure 3-2 Security Functionality Enforced During Object Invocations and Applications

Most of the security services utilize the principal's credentials either at the client (before invoking the target object) or at the target. For example, the ORB security services use these credentials for secure associations, access control, and auditing.

To fit with the standard CSI security mechanisms, user/principal authentication must produce credentials suitable for both client side security controls and to fit with the security mechanisms used for secure invocations. A single credential's object may have security context information for more than one mechanism. Security services at the client application use these credentials to enforce security there.

Access control policies at the target generally depend on the initiating principal's privilege attributes (which generally includes an identity). Normally they rely on information from the credentials being passed from the client to the target. Other access policies may use the pull model for obtaining privileges at the target. For example, an access policy at the target could obtain the access identity using the **get_attributes** function. It could then call, in a non-standard way, on whatever service provides privileges in this case. Alternatively, an attribute Mapper (see Section 3.1.13.3, "Attributes at the Target," on page 3-28) could be used before calling the access policy (if this optional facility is supported).

Audit policies generally require an audit id, though this may be derived like the access id from a single identifier.

This specification allows unauthenticated and authenticated users; however, unauthenticated principals do not have identity attributes or privilege attributes. In the protocols defined here, principals must be authenticated.

The privilege and other attributes, as seen by the **AccessDecision** object at the target, may not be those passed from the client because the security mechanism may have moderated what is available to the object system.

3.1.13.1 Credential Content at the Client

Credentials are made available to the client as the result of authenticating the user (or other principal), though they may be modified later. Authenticated users have two types of attributes visible to applications and relevant to secure interoperability:

1. Privilege attributes used for access control. These include the **AccessId** (the principal's identity as used for access control); other standard CORBA security attributes such as **GroupId**, **Role**, **Clearance**, and enterprise defined attributes.
2. Identity attributes used for purposes other than access control. Only the audit identity is relevant here.

   At CSI levels 0 and 1, the only attributes which must be visible to the client and target are the **AccessId** and **AuditId**. These will normally be the user's security name.

   At CSI level 2, a wider range of privilege attributes is supported.

   - All conformant ORBs can generate (via security services) credentials with the following privilege attributes:
     - **AccessId**
     - **AuditId**
     - **Role**
     - **GroupIds** - a primary group and other groups
   - There may be a single identity (e.g., the access identity) which can also be used for auditing, or separate **AccessId** and **AuditId** may be generated. **AuditId** may be anonymous.
   - Optionally, there may also be other privilege attributes including user defined attributes.

3.1.13.2 Attributes During Transmission

At levels 0 and 1, only the principal's identity is transmitted. No other attributes are transmitted.

At level 2, a wide range of privileges can be transmitted including standard CORBA attributes and optionally user defined ones. Attributes may have individual defining authorities, as at the IDL interface, or share a defining authority.

3.1.13.3 *Attributes at the Target*

At CSI levels 0 and 1, when only a single identity (e.g., the security name) is transmitted, that single identity is used to generate the **AccessId** and the **AuditId** at the target. When using the CSI-ECMA protocol at level 0 or 1, principal identity attributes are transmitted separately from the security name; therefore, the **AccessId** and **AuditId** do not have to be generated from the security name.

At CSI level 2, all conformant ORBs can accept:

- Separate access and audit ids or a single identity used for both purposes.
- Transmission of any privileges defined in Appendix B, Section B.11.1, "Attribute Types," on page B-27, and any privileges with Object Identifiers which can be mapped to **SecurityAttributes**.

This range of privileges can be used in access decisions at the target. Even if these privileges are not used by the invocation access policy to control access to the target object, they may be obtained by the application using **Current::get_attributes** or **Credentials::get_attribute** and used in application access decisions.

The attributes at the target appear as defined in Section 3.6.2.1, "Privilege Attributes," on page 3-68. For example, they have:

- an Attribute type (family definer, family, and the type within this family),
- a defining authority, and
- the attribute value.

The attributes may need to be mapped from their form in transit to the form used at the IDL interface in response to **get_attribute** calls. An attribute mapper may be needed, as shown in Figure 3-3.

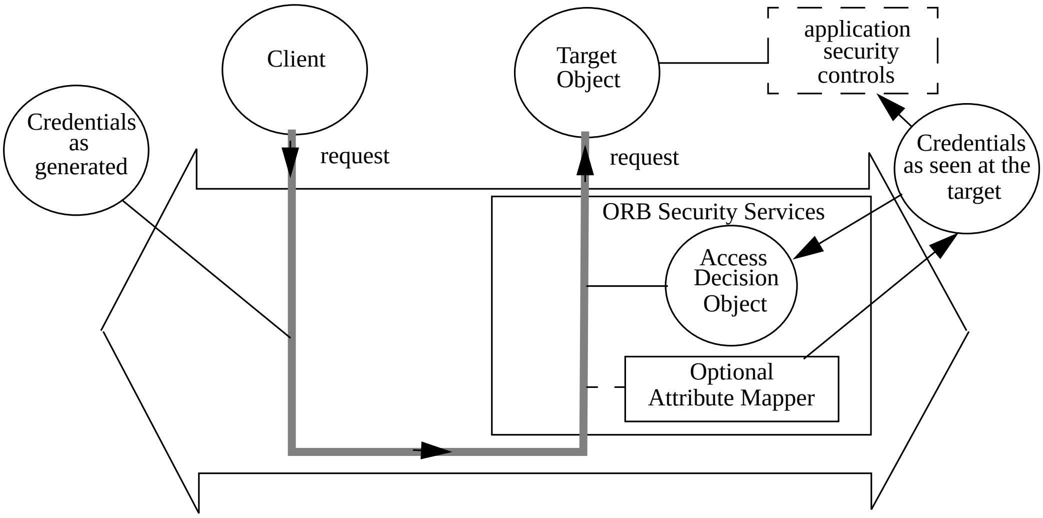


Figure 3-3 Attribute Mapper Diagram

This mapping depends on:

- Which functionality level is supported. At levels 0 and 1, a single name must be mapped to provide both **AccessId** and **AuditId**. This will be the security name if the protocol does not carry a separate **AccessId** or **AuditId**; both the SPKM and GSS-Kerberos protocols use the security name.
- Whether the access control decisions at the target uses attribute values which are valid externally from the ORB/operating system (for example, in a domain of heterogeneous systems), or whether the Access policies use local attributes (such as operating system ids).

  In line with the OMG requirement for portability, externally valid attributes are the norm, and must be supported in conformant ORBs (so that an application which includes administration of its access policy is portable between unlike systems). Mapping to local attributes may also be provided, but is not standardized in this specification.

3.1.13.4 Mapping Security Names to Externally Valid Identities

Where the only client attribute transmitted is the security name, CSI conformant ORBs map this onto both the **AccessId** and **AuditId** in the received credentials. These both have the same value.

When using the GSS-Kerberos, the security name protocol has two components: a realm name and a principal name. The security name is of the form **principal@realm.** The principal name may be a multi-component name with components separated by slash (/) - see [12] section 2.1.1.

When using a public key based mechanism, the security name is a directory name. This is a multi-part name (e.g., country, organization, organization unit, surname, and common name). The security name is returned from the security mechanism in the form of a string complying with [4] for the string representation of distinguished names. The separators between components of the name may be commas or semicolons.

In both cases, the full Security name is used as the value for the **AccessId** and **AuditId** in the IDL SecurityAttributes. This means the form of these attributes are dependent on the security mechanism used, as Kerberos and X.500 names have different forms.

3.1.13.5 Mapping Other Attributes to Externally Valid IDL Attributes

Other security attributes may also be transmitted from the client when using the CSI-ECMA protocol. For example, at level 2, there could be a **Role**, **GroupId**, and enterprise specific attributes as well as **AccessId** and/or **AuditId**. Also, separate **AccessId** and **AuditIds** may be transmitted.

In general, these will already have values which are valid outside a particular ORB and operating system; therefore, the mapping is mainly to put these in the form of an IDL SecurityAttribute. However, if a separate **AuditId** has not been transmitted, the **AuditId** value will be copied from the **AccessId**. Also, if a separate defining authority is not transmitted for an attribute, the defining authority for the attribute in IDL is set from the issuer Domain of the authority who generated the Privilege Attribute Certificate containing the privileges. Note also that the target security policy may restrict which of the attributes are available to the application.

Attribute types in transmission are identified by Object Identifiers. For the standard attribute types such as Role or GroupId (as defined in Appendix B, Section B.11.1, “Attribute Types,” on page B-27), the type is automatically translated to the appropriate CORBA family and attribute type. The value is also re-encoded, if needed, from ASN.1 to the equivalent IDL type.

We propose that OMG should register itself in the ISO Object Identifier space. A **SecurityAttribute** type where there is a family definer registered with OMG (see Appendix B, Section B.11, “Values for Standard Data Types,” on page B-26) can then be transmitted with an Object Identifier of:

<iso>..<omg>.<security>.<family_definer>.<family>.<attribute type>

which then can be mapped automatically onto the CORBA **SecurityAttribute** structure.

Attributes other than the standard attributes and those with CORBA family Object Identifiers are not guaranteed to be understood at the target; therefore, they may not be automatically mapped to CORBA families and types. Such mapping can be done by an optional attribute mapper which understands these attribute types.

3.1.13.6 Mapping to Local Attribute Values

An ORB can support mapping of the security name and other attributes to local operating system values such as UNIX uids and gids. This mapper could generate different **AccessIds** and **AuditIds**. Note that when using local values, the application (particularly the access policy administration) will not be portable to other types of system.

Mapping of these values is specific to the ORB and/or operating system. This standard does not specify how this mapping is done, whether it calls on other software to do it, or what types of values it generates. However, the defining authority in the IDL SecurityAttribute must identify the local environment responsible for the meanings of these values, so the application can determine where these values are valid.

Mapping to local attributes may be done by an optional attribute mapper (see Section 3.1.16.1, “Attribute Mapping,” on page 3-33).

3.1.14 CORBA Interfaces

In this section:

- Profiles of the interfaces defined in sections 2.3 through 2.5 are defined.
- Values of certain IDL constants relevant to these profiles are defined.

- Restrictions that application that use the Security interfaces must adhere to for conforming to this Common Secure Interoperability standard are identified.

3.1.14.1 Service Options for Common Secure Interoperability

The following Service Options are returned by **ORB::get_service_information** representing the level of CSI that is supported by the ORB:

```
module Security {
    const CORBA::ServiceOption CommonInteroperabilityLevel0 = 10;
    const CORBA::ServiceOption CommonInteroperabilityLevel1 = 11;
    const CORBA::ServiceOption CommonInteroperabilityLevel2 = 12;
};
```

The common interoperability protocols supported are identified using a **ServiceDetail** structure with a **ServiceDetailType** of **Security::SecurityMechanismType**, as described in Section 2.3.2, “Finding Security Features,” on page 2-73. The values for the CSI mechanisms are defined in Appendix B, Section B.2, “General Security Data Module,” on page B-1.

3.1.14.2 Mechanism Types

The mechanism at the application interface is defined as **Security::MechanismType** (a string). CSI mechanisms are encoded in the **MechanismType** string by concatenating a mechanism id and zero, one, or more cryptographic profiles separated by commas.

The mechanisms supported by an object are identified by tags in its **IOR**. In the MechanismType, the mechanism is identified by a “stringified” form (e.g., the integer value 123 represented as the string “123”) of the **TAG_x_SEC_MECH** id value for that mechanism. Mechanisms supported by SECIOP based protocols are:

- **SPKM_1** or **SPKM_2**: the level 0 public key mechanisms using the SPKM protocol.
- **KerberosV5**: the level 1 secret key mechanism using GSS Kerberos protocol.
- **CSI_ECMA_Secret**: the CSI-ECMA secret key mechanism, using Kerberos V5.
- **CSI_ECMA_Hybrid**: the CSI-ECMA mechanisms which uses secret key technology for key distribution within a domain, but public key between domains.
- **CSI_ECMA_Public**: the CSI-ECMA public key mechanism.

Cryptographic profiles are identified by a “stringified” form of the **CryptographicProfile** value as used in the **IOR**.

MechanismType is used in a number of operations. These include operations that:

- Deal with the mechanisms and cryptographic profiles in **MechanismsPolicy** object for use with **get_policy** and **set_policy_overrides** on an object reference. In this case, the **mechanisms** attribute of the **MechanismPolicy** object (see

Section 2.3.6.2, “Client Side Invocation Policy Objects,” on page 2-87), contains all the Cryptographic profiles available with that mechanism to communicate with that target.

- Specify a security mechanism to use when talking to a target (e.g., using the **MechanismPolicy** object with the **set_policy_overrides** on an object reference and **Vault::Init_security_context** on the **Vault**). In this case, either just the mechanism name may be specified (in which case, a default cryptographic profile will be used) or a mechanism name and cryptographic profile may be specified.

The **get_service_information** operation on the ORB can also return the mechanism, though in this case, it is in the form of a **sequence<octet>**.

Mechanism tags in the **IOR** and mechanism type Object Identifiers (as in GSS-API) in SECIOP messages are also used as appropriate.

3.1.14.3 Delegation Related Interfaces

Interfaces to handle no delegation, simple delegation, and composite delegation (hence delegation interfaces for CSI levels 0, 1, and part of 2) are defined in Section 2.3.11, “Delegation Facilities,” on page 2-106).

CSI level 2 also supports controls on the delegation of credentials. How to specify these controls is not included in this specification. It is assumed that it is handled by administrative action. For example, it may be done by associating the delegation controls with a user or an attribute set selected when the user logs on or selects attributes at other times. Management of attributes associated with a principal is considered out-of-scope of this specification.

No facilities are currently defined for an application object to specify controls it wishes to apply on delegating its credentials. In future, such facilities may be considered for CORBA Security - see Appendix F, Section F.13, “Advanced Delegation Features,” on page F-5.

3.1.15 Support for CORBA Security Facilities and Extensibility

This CSI specification assumes that the ORB conforms to at least CORBA Security mandatory facilities (except for delegation at CSI level 0), and requires that this functionality can be supported across different ORBs using any of the CSI level specified here.

The CORBA Security specification allows use of a wide range of security policies, facilities, and mechanisms. Conformant ORBs can restrict which of these can be used during interoperability, as follows:

- The protocol may not carry the privileges the target needs for some of its access policies. For example, at CSI levels 0 and 1 only an identity is supported.
- It may not carry the type of audit identity needed for the audit policy. For example, it may not be able to carry an anonymous **AuditId**.

- It may not support composite delegation. (CSI levels 0 and 1 do not; in CSI level 2 it is not mandatory).
- There are restrictions on the SECIOP exchanges (e.g., separate request and response protection is not supported).
- Unauthenticated users may not be supported (All CSI levels).

3.1.16 Security Replaceability for ORB Security Implementors

Security policy implementations could be replaced to provide new security policies as discussed in Section 2.5.3, "Replaceable Security Services," on page 2-168.

This common Interoperability specification affects replaceability in two areas:

1. Mapping of attributes as described in Section 3.1.13, "Model for Use and Contents of Credentials," on page 3-26 affects replaceable security policies which use these attributes.

2. Use of the Generic Security Services API (GSS-API) within the **Vault** and **Security Context** implementation objects described in Section 2.5.2, "Implementation-Level Security Object Interfaces," on page 2-149, should make these objects independent of the particular security mechanisms used.

3.1.16.1 Attribute Mapping

As described in Section 3.1.13.3, "Attributes at the Target," on page 3-28, the form of attributes may need to be mapped before being made available to a target security policy (**AccessPolicy** or **AuditPolicy**) or to the target object.

No interface for an attribute mapper is currently defined; therefore, it is not possible to replace attribute mapping independently of the ORB/security mechanism. Such an interface may be defined in the future.

3.1.16.2 Use of GSS-API

The choice of security mechanism is not visible outside the **Vault** and **Security Context** objects, except for the identification of the **Mechanism** (and associated cryptographic profiles) in the **IOR** and in the **MechanismPolicy** object (see Section 2.3.6.2, "Client Side Invocation Policy Objects," on page 2-87).

The **Vault** and **Security Context** can use GSS-API to implement their security functions, and so remain independent of security mechanism.

If only CSI level 0 or 1 facilities are used, the standard GSS-API interface (as defined in RFC 1508) can be used. If CSI level 2 facilities are needed, this requires use of attributes other than the security name, and may also use delegation controls. Therefore, it requires use of an extended GSS-API, such as [12].

Use of GSS-API is a recommendation, but is not proposed as a conformance option for this CSI specification or for the CORBA Security specification.

3.2 Secure Inter-ORB Protocol (SECIOP)

To provide a flexible means of securing interoperability between ORBs, a new protocol is introduced into the CORBA Interoperability Architecture. This protocol sits below the GIOP protocol and provides a means of transmitting GIOP messages (or message fragments) securely.

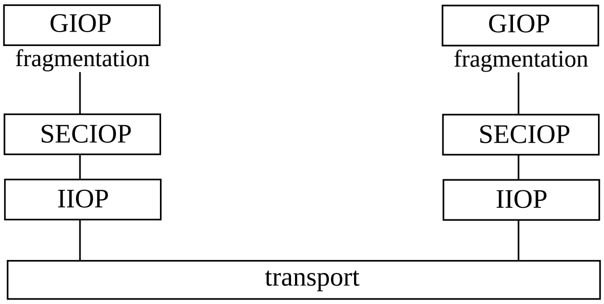


Figure 3-4 Position of SECIOP Protocol

SECIOP messages support the establishment of Security Context objects and protected message passing. Independence from GIOP allows the GIOP protocol to be revised independently of SECIOP (e.g., to support request fragmentation). A synchronized pair of **Security Context** objects and their corresponding sequencing state is called a *security association*.

SECIOP is sub-layered into a Sequencing Layer and Context Management Layer.

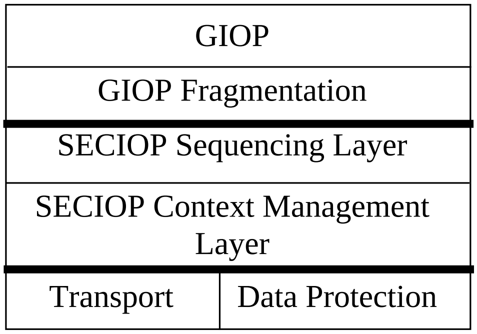


Figure 3-5 Sublayers of SECIOP

This specification assumes that SECIOP provides services to the GIOP Fragmentation Layer. Providing the interface to GIOP fragmentation is the SECIOP Sequencing Layer. It has responsibility to securely and reliably deliver GIOP fragments to the correspondent. It encapsulates GIOP fragments into frames for protection by the SECIOP Context Management Layer. It also uses frames that do not carry fragments to coordinate the distributed sequence number state bound to the security association. SECIOP frames are encoded in CDR and delivered to the SECIOP Context Management Layer.

The SECIOP Context Management Layer accepts frames from the Sequencing layer and encapsulates them in a Context Management message. These messages are cryptographically protected by tokens, which are the product of the Data Protection layer, normally GSSAPI. The Context Management Layer carries Data Protection tokens in SECIOP messages for the purpose of both managing security associations and for securing frames moving between it and the correspondent. The Context Management layer uses the Transport layer to communicate with the correspondent. The Context Management layer is driven by the finite state machine defined in Table 3-4 on page 3-48 and Table 3-5 on page 3-51.

3.2.1 Architectural Assumptions

SECIOP is designed to support a rich variety of different software implementation architectures. In order to operate in the most sophisticated of these, the design assumes both clients and targets are multi-threaded and that a single TCP connection can support multiple security associations.

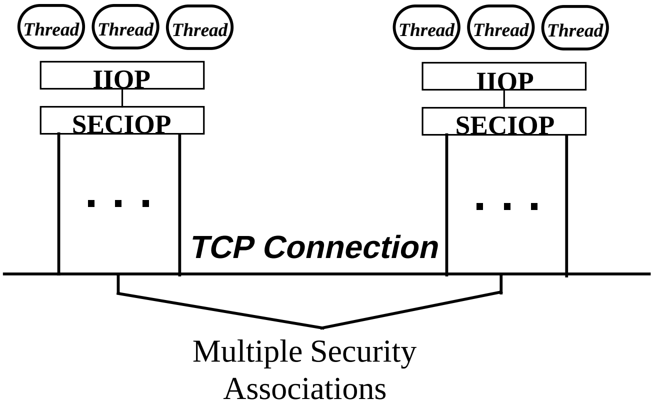


Figure 3-6 Architectural Assumptions

This specification assumes the following environmental and implementation characteristics:

- Each SECIOP secure association is bound to a single transport connection. This ensures that GIOP fragments are not reordered due to thread scheduling anomalies. It also guarantees that a response to a GIOP request returns on the same transport connection as the request, which is required by the GIOP specification.
- SECIOP may use multiple security associations over the same transport connection. This allows implementations to multiplex SECIOP traffic, which can improve performance.
- SECIOP ensures that fragments are sent over transport connections in their sequence number order. This means that once an SECIOP sequence number is assigned to a fragment, the fragment will be processed by the Data Protection layer and sent over transport before any other fragment with a larger sequence number protected by the same security association.

- When a transport connection is closed, all SECIOP secure associations using it are closed as well. This may require discarding fragments on the Sequencing layer retransmission queue that have not yet been acknowledged. This is acceptable, since closing a transport connection forces GIOP to mark any outstanding Requests as MAYBE. Furthermore, closing a transport connection must be visible to both sides of the connection, so both sides of the security association will follow this rule.
- There is always a listener at the client and server prepared to receive and process SECIOP messages. This is necessary, since the loss of security context information by one side or the other requires a re-establishment of the security association. This in turn requires both client and server to be listening for security context management messages.
- Both the client and server may initiate security context establishment (i.e., send the **EstablishContext** message). This is necessary when a server needs to return a response to the client but discovers that the security association is no longer valid (e.g., it has timed out).
- SECIOP sequence numbers should never wrap around to zero. If they did, it would introduce a replay threat. Consequently, when the SECIOP Sequencing Layer receives an acknowledgment to a fragment with a sequence number equal to 1/2 the precision of an unsigned long (the type used for sequence numbers), it must discard the existing security association and establish a new one. This rule derives from the sequencing algorithm property that up to 1/2 of the possible sequence numbers in the higher 1/2 of the sequence number space may be used for new fragments before the fragment associated with the last sequence number in the lower 1/2 of the space is acknowledged. Note that the SECIOP sequencing state should not be discarded when a new security context is established.
- There is Data Protection protocol information (e.g., GSSAPI tokens) carried within SECIOP messages. This protocol should be configured so it does not itself provide sequencing services. Otherwise, there could be interference between the two layers, causing unnecessary lost service.

3.2.2 SECIOP Sequencing Layer

SECIOP sequencing uses a modified data link layer protocol based on one in production at Lawrence Livermore National Laboratory for over 10 years. This protocol, called modified ALP, is described below.

SECIOP Sequencing layer frames are carried in **MessageInContext** messages (see Section 3.2.3.6, “Message Definitions,” on page 3-44). The **message_protection_token** in this message is defined to be an opaque sequence of octets. In order to support sequencing, however, the Sequencing layer defines the structure of these octets as follows (the definition of **MessageInContext** is repeated here for completeness):

```
struct MessageInContext {
    ContextIdDefn       message_context_id_defn;
    TokenType           message_protection_token_type;
    ContextId           message_context_id;
    sequence<octet>     message_protection_token;
};
```

message_protection_token is obtained by processing the frame header encoded in CDR as a **SequencingHeader** followed by the octets of the frame data. The combination of frame header and frame data is called a **SequencedDataFrame**.

The **frame_header** field is always present in a **SequencedDataFrame**; however, the **frame_data** field may or may not be present. If not present, the length of the **MessageInContext** message includes only the octets up to and including the frame header.

The **SequencingHeader** has the following definition:

```
struct SequencingHeader {
    octet               control_state;
    unsigned long       direct_sequence_number;
    unsigned long       reverse_sequence_number;
    unsigned long       reverse_window;
};
```

The **control_state** field contains information necessary for the reliable delivery of frame data between the correspondents. It is encoded as follows (**control_state[x]** is bit **x** in the octet, where bit **0** is the least significant bit):

```
control_state[0] : direct_phase
control_state[1] : direct_fragment
control_state[2] : direct_reply
control_state[3] : reverse_phase
```

3.2.2.1 *Protocol State*

The new version of SECIOP uses a variant of ALP (A Link Protocol) a data link layer protocol. Its design relies on the principal of state-exchange, a coherent design strategy that produces protocols that are easy to understand, clearly documented, and lend themselves to rigorous analysis.

It is assumed that the reader is familiar with this link-layer protocol. Those unfamiliar with it are referred to the paper [18].

The main body of this paper establishes the rationale for the state-exchange model, while Appendix A documents the ALP protocol itself.

To embed ALP within SECIOP, each participant in a security association maintains state used for sequencing. This state is embodied in several variables that the participant manages as well as a queue of data fragments.

These are:

output_queue	A queue of fragments. SECIOP is responsible for securely and reliably moving them to the correspondent.
output_phase	A boolean indicating a stream of transmissions.
output_sequence_number	The sequence number associated with the oldest fragment on **output_queue**.
output_count	The number of fragments in **output_queue** that have been transmitted but not yet accepted or rejected.
output_window	The window size for output fragments.
output_length	The length of the **output_queue**.
input_phase	The phase expected with the next input fragment.
input_sequence_number	The sequence number expected for the next fragment.
input_window	The window size for input fragments.
input_reply	A boolean, which if set indicates at least one frame should be sent.

In addition to these variables, the SECIOP Sequencing layer has available the following functions and procedures:

receive*()*	Returns a received frame.
newframe*()*	Returns an empty frame buffer (i.e., a **SequencedDataFrame** struct).
send(f)	Sends the frame **f**.
discard(f	Discards the frame **f**.
pop(q*)*	Removes and discards the leading element **q[0]** of the queue **q**. The index of the remaining elements is decremented by one.
forward(d*)*	Forwards the fragment **d** to the GIOP fragmentation layer.
mod(n,m)	Returns the remainder from the division of the integer **n** by the positive integer **m**.
min(n_1,n_2,..., n_x)	Returns the smallest of the integers $\mathbf{n_1}$ through $\mathbf{n_x}$.
resync()	Signals the SECIOP Context Management layer to discard the old security association bound to the sequencing state and establish a new one. [NB: this is not included in the original ALP definition, since the notion of a security context is not germane to its original purpose].
frame_data(f)	The **frame_data** field of a **SequencedDataFrame** message **f**.

Finally, the value **M** is defined to be the number of values that can be carried by an **unsigned long**.

3.2.2.2 Protocol Initialization

The next three sections describe the operation of the Sequencing layer. The algorithms are expressed in a pseudo-ALGOL syntax (with slight modifications from the C programming language to facilitate writing conditional expressions).

When the GIOP fragmentation layer requests the transport of a fragment to a destination for which no SECIOP secure association exists, the SECIOP Sequencing layer creates a state record consisting of the variables defined in the last section and initializes them as follows:

output_queue := empty;

output_phase := 0;

output_sequence_number := 0;

output_count := 0;

output_window := 0;

output_length := 0;

input_phase := 0;

input_sequence_number := 0;

input_window := [an implementation defined value < M/2];

input_reply := 1;

In the original definition of ALP, the initial values of some of these variables was unspecified. This specification defines these initial values so that there need be no handshaking activity between the correspondent's SECIOP Sequencing layer code in order to move the first fragment. This facilitates transaction style operations in which a security association is established without mutual authentication, thus allowing the first fragment to be sent without waiting for an SECIOP reply.

Another slight change from the original definition of ALP is the requirement that the window size must never be set greater than (M/2)-1. This restriction is necessary so that two acknowledgments carrying equal sequence numbers referring to different fragments are never protected using the same security context. Without this restriction, there is a hazard that an intruder could replay an acknowledgment to a fragment not received, thereby causing the fragment to be dropped.

Once a security context is established, the SECIOP Sequencing layer processes the information in a SequencedDataFrame according to the algorithms given in the next two sections.

3.2.2.3 Upon Receipt of a SequencedDataFrame

Note – This text is taken directly from the cited paper and slightly modified to adapt it to using security contexts. The code that has been modified is called out by a solid black line on the left side.

The receiver code below is called on both the target and client sides when the SECIOP Finite State Machine (FSM) is in state S3 and a **MessageInContext** arrives.

```
begin comment This algorithm should be executed after receipt of each non-
            erroneous frame;
f := receive();
if direct_sequence_number(f) == input_sequence_number
    and
  direct_phase(f) == input_phase
then
    if direct_fragment(f) == 1 and input_window > 0
    then comment An input fragment has arrived in sequence. Accept it;
        input_sequence_number := mod(input_sequence_number + 1, M);
        forward(frame_data(f));
    fi;
else comment An input fragment has been lost. Prepare to accept retransmissions;
    input_phase := 1 - direct_phase(f);
fi;
if mod(reverse_sequence_number(f)-output_sequence_number, M) <= output_count
then comment The received reverse sequence number is not anomalous;
    while reverse_sequence_number(f) != output_sequence_number
        do comment Discard accepted output fragments;
            pop(output_queue);
            output_sequence_number := mod(output_sequence_number + 1, M);
            if mod(output_sequence_number, M/2) == 0
            then comment all fragments up to and including (M/2)-1 have been
                acknowledged. Use a new security context for future fragments
                to avoid replays. Resynchronizing the security context when
                exactly half of the sequence number space has been "used"
                achieves two objectives : 1) it ensures that no two fragments with
                the same sequence number are protected by the same security
                context, and 2) it ensures that two acknowledgments carrying the
                same sequence  number, but acknowledging different fragments
                are not protected using the same security context. The latter
                objective requires the further limitation that the window size is
                never set greater than (M/2)-1;

            resync();
            fi;
        output_count := output_count - 1;
        output_length := output_length - 1;
        od;
    output_window := reverse_window(f);
fi;

if reverse_phase(f) != output_phase
then comment Prepare to retransmit rejected output packets;
    output_phase := reverse_phase(f);
    output_sequence_number := reverse_sequence_number;
    output_count := 0;
fi;

if direct_reply(f) == 1 or output_length > 0
then comment State is unsatisfactory;
    input_reply := 1;
fi;
discard(f)
end;
```

3.2.2.4 Sending a SequencedDataFrame

This sending code is called on the target and client side when the Sequencing Layer caller has a fragment to send. Certain events within the Sequencing layer also cause this algorithm to be executed. Specifically, the sending algorithm is executed when the receiving code in the previous section is executed and a non-erroneous frame is received. Also, **input_reply** should be set to 1 and the sending code executed:

1. when an erroneous frame is received;
2. when a new security context is established;
3. when an **EstablishContext** message is sent with messages allowed;
4. when **input_window** is changed by the implementation; and
5. upon initialization of the Sequencing state.

```
begin
```

```
        while output_count < min(output_window, output_length) or input_reply == 1
            do comment A frame should be sent;
                f := newframe();
                input_reply := 0;
                direct_phase(f) := output_phase;
                direct_sequence_number(f) := mod(output_sequence_number +
                                                 output_count, M);
                if output_count < min(output_window, output_length)
                then comment A fragment could be included in the frame.
                    direct_fragment(f) := 1;
                    frame_data(f) := output_queue[output_count];
                    output_count := output_count + 1;
                fi;
                if output_length > 0)
                then comment Not all packets have as yet been accepted;
                    direct_reply(f) := 1
                fi;
                reverse_phase(f) := input_phase;
                reverse_sequence_number(f) := output_sequence_number;
                reverse_window(f) := output_window;
                send(f);
            od
    end
```

3.2.3 SECIOP Context Management Layer

The SECIOP Context Management Layer establishes and controls a secure association between a client and target. It also provides a means for the transmission of protected messages between clients and targets.

3.2.3.1 SECIOP Context Management Layer Message Header

SECIOP Context Management messages share a common header format with GIOP messages defined in the *Common Object Request Broker: Architecture and Specification*. The fields of this header have the following definition.

- **magic** - identifies the protocol of the message. Each protocol (GIOP, SECIOP) is allocated a unique identifier by the OMG. The value for SECIOP is “SECP.”
- **protocol_version** - this contains the major and minor protocol versions of the protocol identified by magic. The value for the version of SECIOP defined here is 1 major version, 1 minor version. This field is called **GIOP_version** in **GIOP::MessageHeader_1_1**.

- **byte_order** - as in the GIOP header definition.
- **message_type** - this is the protocol specific identifier for the message.
- **message_size** - as in the GIOP header definition.

3.2.3.2 SECIOP Context Management Layer Protocol

Where possible, SECIOP Context Management messages are sent with GIOP messages rather than as separate exchanges. However this is not always possible (e.g., when the client wishes to authenticate the target before it is prepared to send a GIOP message).

The SECIOP Context Management Layer has the following message types:

```
module SECIOP
    enum MsgType {
        MTEstablishContext, MTCompleteEstablishContext,
        MTContinueEstablishContext, MTDiscardContext,
        MTMessageError, MTMessageInContext
    };

    typedef unsigned long long ContextId;

    enum ContextIdDefn {
            CIDClient,
            CIDPeer,
            CIDSender
    };

    enum ContextTokenType {
            SecTokenTypeWrap,
            SecTokenTypeMIC
    };
};
```

3.2.3.3 ContextId

This type is used to define the identifiers allocated by the client and target for the association.

3.2.3.4 ContextIdDefn

This enum is used to define the kind of context identifier held in an SECIOP message. The context identifier will either be the one specified by the client which established the context or it will be the identifier associated with the receiver of the message (i.e., the request target for request or request fragment messages or the request client for reply or reply fragment messages). The value must equal Client if the value of **target_context_id_valid** in the **CompleteEstablishContext** was false or the message has not yet been exchanged. It must equal Peer if the value of

target_context_id_valid in the **CompleteEstablishContext** was true. The use of peer identifiers allows the recipient of the message to more efficiently find its security context. The values are defined as:

- **CIDClient** - the context id is that of the association's client.
- **CIDPeer** - the context id is that of the recipient of the message.
- **CIDSender** - the context id is that of the sender of the message. This is only used with the **DiscardContext** message when the sender of the **DiscardContext** message has no context and has received a message which it cannot process.

3.2.3.5 *TokenType*

This type is used to indicate the type of **message_protection_token** carried by a **MessageInContext** message. The value **SecTokenTypeWrap** indicates the token was returned by a **GSS_Wrap()** call, while the value **SecTokenTypeMIC** indicates the token was returned by a **GSS_GetMIC()** call.

3.2.3.6 *Message Definitions*

EstablishContext

This message is passed by the client to the target when a new association is to be established. Its definition is:

```
struct EstablishContext {
    ContextId           client_context_id;
    sequence <octet>    initial_context_token;
};
```

- **client_context_id** - this is the client's identifier for the security association. It is passed by the target to the client with subsequent messages within the association. It enables the client to link the message with the appropriate security context.
- **initial_context_token** - this is the token required by the target to establish the security association. It contains a mechanism version number, mech type identifier and mechanism specific information required by the target to establish the context. It may be sent with a protected message (for example if the client does not wish to authenticate the target).

CompleteEstablishContext

This message is returned by the target to indicate that the association has been established. It is sent as a reply to an establish context or continue establish context. It may be sent with a GIOP reply or reply fragment. Its definition is:

```
struct CompleteEstablishContext {
    ContextId           client_context_id;
    boolean             target_context_id_valid;
    ContextId           target_context_id;
    sequence <octet>    final_context_token;
};
```

- **client_context_id** - this is the client's identifier for the security association. It is returned by the target to the client to enable the client to link the message with the appropriate security context.
- **target_context_id_valid** - this indicates whether the target has supplied a **target_context_id** for use by the client. True indicates that the following field is valid.
- **target_context_id** - the targets identifier for the association. It is passed by the client to the target with subsequent messages. It enables the target to associate a local identifier with the context to allow the target to identify the context efficiently.
- **final_context_token** - this is the token required by the client to complete the establishment of the security association. It may be zero length.

ContinueEstablishContext

This message is used by the client or target during context establishment to pass further messages to its peer as part of establishing the context. It may be the response to an establish context or to another continue establish context. It is defined as:

```
struct ContinueEstablishContext {
    ContextId           client_context_id;
    sequence <octet>    continuation_context_token;
};
```

- **client_context_id** - the client's identifier for the association. It is used by both client and target to identify the association during the establishment sequence.
- **continuation_context_token** - this is the security information required to continue establishment of the security association.

DiscardContext

This message is used to indicate to the receiver that the sender of the message has discarded the identified context. Once the message has been sent the sender will not send further messages within the context. The message is used as a hint to enable contexts to be closed tidily. Its definition is:

```
struct DiscardContext {
    ContextIdDefn       message_context_id_defn;
    ContextId           message_context_id;
    sequence <octet>    discard_context_token;
};
```

- **message_context_id_defn** - the type of context identifier supplied in the **message_context_id** field.

- **message_context_id** - the context identifier to be used by the recipient of the message to identify the context to which the message applies.
- **discard_context_token** - optional token provided by the sender to assist the receiver in cleaning up its security context state.

MessageError

This message is used to indicate an error detected in attempting to establish an association either due to a message protocol error or a context creation error. The message is also used to indicate errors in use of the context.

```
struct MessageError {
    ContextIdDefn       message_context_id_defn;
    ContextId           message_context_id;
    long                major_status;
    long                minor_status;
};
```

- **message_context_id_defn** - the type of context identifier supplied in the **message_context_id** field.
- **message_context_id** - the context identifier to be used by the recipient of the message to identify the context to which the message applies. It is either the client's identifier for the context (type client) or the receiver of the messages identifier (type peer).
- **major_status** - the reason for rejecting the context. The values used are those defined by the GSS API (RFC 1508) for fatal error codes.
- **minor_status** - this field allows mechanism specific error status to further define the reason for rejecting the context. It is not defined further here.

MessageInContext

Once established messages are sent within the context using the **MessageInContext** message. Its definition is:

```
struct MessageInContext {
    ContextIdDefn       message_context_id_defn;
    TokenType           message_protection_token_type;
    ContextId           message_context_id;
    sequence <octet>    message_protection_token;
};
```

- **message_context_id_defn** - the type of context identifier supplied in the **message_context_id** field.
- **message_protection_token_type** - indicates whether the **message_protection_token** is a **SecTokenTypeWrap** or **SecTokenTypeMIC** token.
- **message_context_id** - the context identifier to be used by the recipient of the message to identify the context to which the message applies.

- **message_protection_token** - the sign or seal token for the message. This is a self defining token which indicates how the message is protected. If the message is not protected the token will be zero length.

For signed and unprotected messages, the **MessageInContext** message is followed by the higher level protocol message being transmitted within a security context (i.e. GIOP message or message fragment). The length of the higher level protocol message is included in the length of the **MessageInContext** message. For sealed messages the length of the higher level protocol message is zero.

3.2.4 SECIOP Context Management Finite State Machine Tables

Table 3-4 on page 3-48 and Table 3-5 on page 3-51 present the state transition rules for the Context Management Layer of SECIOP. The state transitions given in these tables are intended to operate in an environment satisfying the following assumptions:

- Each FSM is associated with a unique pair of principals. When an SECIOP message arrives it is delivered to the FSM associated with the principal from which the message was sent and to which the message is delivered.
- There always exists a sequencing state machine (SSM) in the initialized state with an FSM in state 0 at each end of a TCP connection for those principal pairs without an active SSM/FSM.
- Each SSM is associated with exactly one FSM at a time, although an SSM may be associated with multiple FSMs during its lifetime.
- Each TCP connection can be associated with multiple SSMs.
- Each FSM is associated with exactly one **ContextId** during its lifetime.

3.2.4.1 SECIOP Context Management Protocol State Tables

Note that some mechanisms may start in state S3.

Table 3-4 SECIOP Context Management Finite State Machine -Table 1

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
EstablishContext arrives	**If** create context = OK & context complete, Send CompleteEstablishContext. input_reply := 1. Execute send algorithm. S3. **Else if** create context = OK & context incomplete. Send ContinueEstablishContext. S2. **Else** Send MessageError. Terminate SSM. Terminate.	[Target sent EstablishContext at same time Client did. Client's has precedence] S1.	[Target sent EstablishContext at same time Client did. Client's has precedence] S2.	[Target discarded context without telling client] Create a new FSM in state S0. Deliver EstablishContext message to it. Terminate.
CompleteEstablishContext arrives	[A CompleteEstablish-Context arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	Complete context with target's context id. **If** OK, S3. **Else**, send MessageError. Terminate SSM. Terminate	Complete context with target's context id. **If** OK, input_reply := 1. Execute send algorithm. S3. **Else**, send MessageError. Terminate SSM. Terminate	[A CompleteEstablish-Context arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate

Table 3-4 SECIOP Context Management Finite State Machine -Table 1 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
ContinueEstablish-Context arrives	[A ContinueEstablish-Context arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	[A ContinueEstablishContext arriving in S1 is illegal] Send MessageError. Terminate SSM. Terminate	update context state. **If** OK & context complete, Send CompleteEstablishContext. input_reply := 1. Execute send algorithm. S3. **Else If** OK & context incomplete, Send ContinueEstablishContext. S2. **Else**, Send MessageError. Terminate SSM. Terminate	[A ContinueEstablish-Context arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate
MessageError arrives	[A MessageError arriving in S0 is illegal] Terminate SSM. Terminate	Terminate SSM. Terminate	Terminate SSM. Terminate	[target had trouble using its security context and couldn’t reestablish it] Terminate SSM. Terminate.
Send Frame [Normal send case.]	**If** create context = OK, Send EstablishContext message. **If** Message allowed, Send the frame. S1. **Else** S2. **Else** Terminate SSM. Terminate	Send the frame. S1.	S2.	**If** context valid, Send the frame. S3. **Else** Create a new FSM in state S0. Attach it to SSM. Deliver SendFrame to FSM Terminate

Table 3-4 SECIOP Context Management Finite State Machine -Table 1 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
MessageInContext arrives [Normal receive case.]	[Client has discarded context, but target doesn't know it.] Send DiscardContext. S0	[MessageInContext arriving in state S1 is illegal]] Send MessageError. Terminate SSM. Terminate	[MessageInContext arriving in state S2 is illegal]] Send MessageError. Terminate SSM. Terminate	**If** message OK, Execute receive algorithm. **Else If** context timed out, Send DiscardContext. Create a new FSM in state S0. Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate. **Else If** message bad, but context OK, drop message. input_reply := 1. Execute send algorithm. **Else** Send MessageError. Terminate SSM. Terminate.
DiscardContext arrives	[ignore] S0	[Target doesn't want to create a security association] Terminate SSM. Terminate	[Target doesn't want to create a security association] Terminate SSM. Terminate	[target's context is no longer valid] Create a new FSM in state S0. Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate.
Resync Requested	[ignore. Resync will occur on next SendFrame request] S0	Terminate SSM. Terminate	Terminate SSM. Terminate	Send DiscardContext. Create a new FSM in state S0. Attach it to SSM. Execute send algorithm. Terminate.

Table 3-5 SECIOP Context Management Finite State Machine - Table 2

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
EstablishContext arrives	**If** create context = OK & context complete, Send CompleteEstablishContext. input_reply := 1. Execute send algorithm. S3. **Else if** create context = OK & context incomplete, Send ContinueEstablishContext. S2. **Else** Send MessageError. Terminate SSM. Terminate	[illegal state at Target Side]	[Client wants to start over. Always allow this.] discard partial context. Create a new FSM in state S0. Deliver EstablishContext frame to it. Terminate.	[Client discarded context without telling target.] Create a new FSM in state S0. Deliver EstablishContext frame to it. Terminate.
CompleteEstablishContext arrives	[A CompleteEstablishContext arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	[illegal state at Target Side]	Complete context with context id. **If** OK, input_reply := 1. Execute send algorithm. S3. **Else**, send MessageError. Terminate SSM. Terminate	[A CompleteEstablishContext arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate

Table 3-5 SECIOP Context Management Finite State Machine - Table 2 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
ContinueEstablishContext arrives	A ContinueEstablishContext arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	[illegal state at Target Side]	update context state. **If** OK & context complete, Send CompleteEstablishContext. input_reply := 1. Execute send algorithm. S3. **Else If** OK & context incomplete, Send ContinueEstablishContext. S2. **Else**, Send MessageError. Terminate SSM. Terminate	[A ContinueEstablishContext arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate
MessageError arrives	[A MessageErrort arriving in S0 is illegal] Terminate SSM. Terminate	[illegal state at Target Side]	Terminate SSM. Terminate	[target had trouble using its security context and couldn't reestablish it] Terminate SSM. Terminate.
Send Frame [Normal send case.]	**If** create context = OK, Send EstablishContext message. S2. **Else** Terminate SSM. Terminate	[illegal state at Target Side]	S2.	**If** context valid Send the frame (if not already sent). S3. **Else** Create a new FSM in state S0. Attach it to SSM. Deliver SendFrame to FSM Terminate.

Table 3-5 SECIOP Context Management Finite State Machine - Table 2 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
MessageInContext arrives [Normal receive case.]	[Target has discarded context, but client doesn't know it.] Send DiscardContext. S0	[illegal state at Target Side]	[MessageInContext arriving in state S2 is illegal]] Send MessageError. Terminate SSM. Terminate	**If** message OK, Execute receive algorithm. **Else If** context timed out, Send DiscardContext. Create a new FSM in state S0 Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate **Else If** message bad, but context OK, drop message. input_reply := 1. Execute send algorithm. **Else** Send MessageError. Terminate SSM. Terminate
DiscardContext arrives	[ignore] S0	[illegal state at Target Side]	[Client doesn't want to create a security association] Terminate SSM. Terminate	[client's context is no longer valid.] Create a new FSM in state S0. Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate.
Resync Requested	[ignore. Resync will occur on next SendFrame request] S0	[illegal state at Target Side]	Terminate SSM. Terminate	Send DiscardContext. Create a new FSM in state S0. Attach it to SSM. Execute send algorithm. Terminate.

3.3 The SECIOP Hosted CSI Protocols

All the SECIOP hosted Common Secure Interoperable (CSI) protocols and mechanisms use common elements as far as possible.

- All mechanisms use IOR tags of the form **TAG_x_SEC_MECH** as defined in Section 3.1.4.1, "Security Components of the IOR," on page 3-8.
- The component data structure associated with these tags is common for all protocols and mechanisms in this specification.
- Cryptographic profiles are defined in all cases which allow use of relevant algorithms for confidentiality, integrity, etc. Different mechanisms support some of the same algorithms and one way functions.
- The **MechanismType** as seen at the IDL interface also reflect the mechanism ids and cryptographic profile values in the **IOR** tags.
- Privilege attributes when CSI level 2 is used are the same whether a secret or public key mechanism is used.
- The basic SECIOP token format and some details (such as token types and ids) are common for all protocols.
- All tag components must be encapsulated using CDR encoding.

These protocols are designed to allow use of GSS-API mechanisms. Use of level 2 facilities such as handling of privileges, as defined in Appendix B, Section B.11, "Values for Standard Data Types," on page B-26, imply use of an extended GSS-API such as [23].

3.3.1 IOR

The IOR **TAG_INTERNET_IOP** profile contains the security tags needed for common secure interoperability using GIOP/IIOP. These security tags may be shared with other (non IIOP) protocols, including DCE-CIOP.

The security tags describe what the security target supports and requires, and any mechanism specific data required for secure interoperability using this mechanism.

For common secure interoperability and for all CSI mechanisms and protocols, the IOR must contain at least one appropriate **TAG_x_SEC_MECH** tag.

The **IOR** may also contain the following tags, as defined in "Security Components of the IOR" on page 3-8:

- **TAG_SEC_NAME** provides the security name and may be shared between mechanisms which use the same form of name. Conformant implementation must be able to accept security names shared between such mechanisms.
- **TAG_ASSOCIATION_OPTIONS** may be shared between mechanisms.
- **TAG_GENERIC_SEC_MECH** whose component definition includes a **sequence <TaggedComponents>** includes a **security_mechanism_type** and can include a security name and association options.

If a mechanism is selected for use, and has a defined security name and/or association option, these values are used in preference to any values defined at the higher level. If no name or association options are defined for the mechanism, then the values of these tags in the IIOP profile are used.

3.3.2 Mechanism Tags

The **TAG_x_SEC_MECH** tags for all the CSI mechanisms defined in this specification have an associated component data structure of the same form:

```
struct <mechanism name> {
    AssociationOptions                  target_supports;
    AssociationOptions                  target_requires;
    sequence <CryptographicProfile>     crypto_profiles;
    sequence <octet>                    security_name;
};
```

Names for the CSI mechanisms are:

SPKM_1
SPKM_2
KerberosV5
CSI_ECMA_Secret
CSI_ECMA_Hybrid
CSI_ECMA_Public

Tag ids for the mechanisms are:

TAG_SPKM_1_SEC_MECH
TAG_SPKM_2_SEC_MECH
TAG_KerberosV5_SEC_MECH
TAG_CSI_ECMA_Secret_SEC_MECH
TAG_CSI_ECMA_Hybrid_SEC_MECH
TAG_CSI_ECMA_Public_SEC_MECH

- The association options required/supported by the target are defined in Section 3.3.3, “Association Options,” on page 3-55.
- The sequence of **crypto_profiles** defines one or more cryptographic profile supported by this target using this mechanism as defined in Section 3.3.4, “Cryptographic Profiles,” on page 3-56.
- The security name is defined in Section 3.3.5, “Security Name,” on page 3-57.

3.3.3 Association Options

With all CSI protocols and mechanisms, a secure ORB supporting a target object must be able to put in the IOR any or all of the association options defined in the CORBA Security specification, as required by the target.

All compliant secure ORBs supporting clients must be able to accept all the **target_supports** and **target_requires** association options, and act on these correctly, as defined in “TAG_ASSOCIATION_OPTIONS” on page 3-9.

Two of the association options are replay and misordering detection. While all the protocols in this specification include facilities to detect replay and misordering, in a multi-threading CORBA environment, the calls on the security mechanism are not guaranteed to be made in the same order that the messages they are protecting are transmitted. The facilities in the security mechanisms cannot guarantee that they will correctly detect replay and misordering. An extension to SECIOP is expected in future to provide these checks. Until this change to SECIOP has been specified and adopted (although these association options may be set) replay and misordering detection is not a mandatory part of this specification.

If no association options are specified in the **IOR**, a CSI defined default is assumed.

3.3.4 Cryptographic Profiles

Cryptographic algorithms are used for

- integrity and confidentiality protection of messages,
- establishing the security association between client and target (including peer authentication and establishing session keys),
- deriving dialogue keys for message protection (both confidentiality and integrity), and
- protecting systems security data such as **PACs** (Privilege Attribute Certificates).

The security mechanisms defined here allow a choice of algorithms which can be used for the different functions, depending on

- the needs of the functions, and
- the requirements for international deployment in countries which constrain how cryptography can be used and exported from countries where use of cryptography is controlled.

In some cases, export controls may require international versions of products to use shorter key lengths; therefore, a large number of combinations of algorithms and key lengths may be possible. For interoperability, both client and target must support the same algorithms and key lengths for these functions.

This specification defines a number of *cryptographic profiles*, where each profile identifies a set of algorithms with specified key lengths used by a mechanism for specified functions.

For example, the CSI-ECMA protocol defines a **NoDataConfidentiality** cryptographic profile which can use DES and RSA for protecting the security mechanism, but does not encrypt the ORB request/reply. (The profile for full security would use DES/64 for data confidentiality.)

Cryptographic profiles are identified by a value, represented in **IORs** as an unsigned short:

typedef unsigned short CryptographicProfile;

3.3.4.1 Key Establishment Algorithms

The algorithms used to establish the cryptographic session keys during security associations depend on the type of mechanism.

- Where the secret key (Kerberos based) mechanism is used, either via the GSS Kerberos or CSI-ECMA protocol, the DES algorithm is used.
- When a public key mechanism is used, either via SPKM or CSI-ECMA protocol, the RSA algorithm is used.

3.3.4.2 Common Message Protection Algorithms

Even if different mechanisms and algorithms are used for key establishment, the same algorithms can be used for message protection.

- All CSI mechanisms have cryptographic profiles which include an MD5 hash of the data for integrity, though the hash, in some profiles may be signed or encrypted.
- All CSI mechanisms can use DES in CBC mode for message confidentiality.

3.3.4.3 Cryptographic Profiles Supported by CSI Protocols

A number of cryptographic profiles are defined for each CSI protocol. Further cryptographic profiles using different algorithms can be used with these protocols, but these are not part of this interoperability standard. A target may support several cryptographic profiles for a particular mechanism.

In all cases, support of a CSI protocol requires support for a cryptographic profile which provides integrity of user data, but not confidentiality, as such a profile is easier to deploy internationally. For example, the GSS Kerberos protocol always supports its *MD5* cryptographic profile. Other profiles may also be supported.

3.3.5 Security Name

The form of the security name depends on the security mechanism used. For example, it can be a Kerberos name or a Directory style name. Directory names conform to the string representation defined in [4].

The security name may be at the component level of the **IOR** or higher if shared between mechanisms. If a security mechanism tag, but no security name is present in the **IOR**, the **IOR** is improperly formatted and a CORBA::INV_OBJREF exception shall be raised when the **IOR** is used to specify the target of an operation.

3.3.6 Security Administration Domains

As defined in Section 2.1.8, “Domains,” on page 2-21, a security policy domain is a set of objects to which a security policy applies for a set of security related activities and is administered by a security authority.

Security mechanisms are concerned with the security domains where users and other principals are administered, often by on-line authorities such as Authentication and Privilege Attribute Services. Often, this domain will be the enclosing domain encompassing secure invocation, access control, and other policy domains.

Note that some authorities may be off-line. For example, the Certification Authority used to issue certificates is often off-line.

The security mechanisms specified in this document allow requests to cross domain boundaries. At the boundary, trust between the domains needs to be established. (The way this is done depends on the mechanism used.) Also, the scope of privileges may not always cross the domain boundary. This specification does not define how privileges are mapped on crossing domain boundaries, as this does not affect the protocol.

While all security mechanisms here include the concept of such domains, in Kerberos (used here as the secret key mechanism) these are known as realms. In this specification, the term realm is used in tokens using this mechanism.

3.3.7 Mapping of Common Elements to the SECIOP Protocol

The SECIOP protocol includes the tokens for context establishment and management and per message tokens.

The context establishment tokens contain:

- Information associated with a principal, including at least an identity. (At CSI Level 2, there may be a range of privileges and a separate audit identity, if required.)
- Associated delegation information. Only simple delegation is mandatory to conform to this specification.
- Security information used to establish the client-target object security association.
- Security information used to establish the keys for message protection.

3.3.7.1 Basic Token Format

SECIOP messages include context and message protection tokens.

All CSI mechanisms are usable inside and outside the object environment. In line with standard practice outside the object environment, tokens are defined in ASN.1. and encoded for transmission using BER (in some cases, constrained to the DER subset of these). The token appears as a **sequence<octet>** in CDR encoded SECIOP messages.

These tokens are enclosed within framing as follows:

```
[APPLICATION 0] IMPLICIT SEQUENCE {
        thisMech   MechType
                                -- MechType is OBJECT IDENTIFIER
        innerContextToken ANY DEFINED BY thisMech
                                -- contents mechanism-specific;
        }
```

Note – For conformance to GSS-API, only the initial context token has to use this token framing; however, in the CSI protocols, it applies to all tokens.

The initial context token should include a mechanism version, as well as type. For CSI mechanisms, version numbers are in the mechanism specific information such as the Kerberos ticket or CSI-ECMA **PAC**.

3.3.7.2 Inner Context Tokens

The same token types are used in the different CSI protocols, though not all protocols support all token types. The token types are defined below showing the relationship with GSS-API calls, as all CSI protocols can be implemented using GSS-API.

The inner context tokens used for security association establishment are:

InitialContextToken	Sent by the initiator to a target, to start establishment of a security association in an SECIOP **EstablishContext** message. The token id is **01 00 (hex)**. If GSS-API is being used, it is the value returned by the **GSS_Init_sec_context** call.
TargetResultToken	Sent to the initiator by the target to complete establishment of the context in an SECIOP **CompleteEstablishContext** message. The token id is **02 00 (hex)**. It is returned by **GSS_Accept_sec_context**.
ContinueEstablishToken	Sent either by the initiator or the target to continue context establishment in an SECIOP **ContinueEstablishContext** message. The token id is **03 00 (hex)** (in SPKM) It is returned by either the **GSS_Init_sec_context** call or the **GSS_Accept_sec_context** call.
ErrorToken	Sent on detection of an error during security association establishment in an SECIOP **CompleteEstablishContext** or **ContinueEstablishContext** message. The token id is **03 00 (hex)** (except in SPKM where it is **04 00 (hex)**). It is returned by either the **GSS_Init_sec_context** call or the **GSS_Accept_sec_context** call.

The inner context token for message protection is the **message_protection_token** in the SECIOP **MessageInContext** message. This can take one of the following forms:

MICToken	Sent either by the initiator or the target to verify the integrity of the user data sent in the following GIOP message (or message fragment). The token id is **01 01 (hex)** It is returned by **GSS_GetMIC**.
WrapToken	Sent either by the initiator or the target. Encapsulates the input user data (optionally encrypted) along with integrity check values. The token id is **02 01 (hex)**. It is returned by **GSS_Wrap**.

This specification always uses **MIC** tokens for integrity and **Wrap** tokens for confidentiality. This may ease national use and export problems where only **MIC** tokens are supported.

The inner context token in the **DiscardContext** SECIOP message may optionally contain a **DeleteContextToken**.

ContextDeleteToken	Sent either by the initiator, or the target in an SECIOP **DiscardContext** message to release a Security Association. It is returned by **GSS_Delete_sec_context**.

3.3.8 CSI Protocols

This specification includes three protocols for different circumstances, as described in Section 3.1.6, “Key Distribution Types,” on page 3-13.

In all cases, the appropriate section specifies the cryptographic profiles supported, and the contents of the SECIOP security tokens.

In all cases, the protocol as supported by OMG is a subset of the protocol defined in the source document. For example, in all protocols, channel bindings as defined in GSS-API (and specified in the underlying protocols) are not supported. This is because IP addresses cannot be trusted in current implementations; IP addresses are spoofable. Including the channel binding information would lead to a false sense of security about the source of the transmission.

The protocols described in this specification include SPKM, GSS Kerberos, and CSI-ECMA.

3.3.8.1 SPKM Protocol

The SPKM protocol supports CSI level 0. This is a public key based protocol. The only client information transmitted is its security name. See Section 3.4, “SPKM Protocol,” on page 3-61.

3.3.8.2 GSS Kerberos Protocol

The GSS Kerberos protocol supports CSI level 1. This is a secret key based protocol. The only client information transmitted is its security name. See Section 3.5, "GSS Kerberos Protocol," on page 3-64.

3.3.8.3 CSI-ECMA Protocol

The CSI-ECMA protocol also supports the privilege handling, separate **Auditid**, and delegation controls of CSI level 2. Subschemes within this protocol support the three key distribution options: secret, public, and hybrid. See Section 3.6, "CSI-ECMA Protocol," on page 3-67 for additional information.

To support this flexibility, the **initial_context_token** is split into three parts; therefore, the attributes for access control are independent of the key distribution method, and this is independent of the cryptography used for message protection. The token contains:

- Authorization information - attributes of a principal are held in a Privilege Attribute Certificate (PAC) with any associated information needed for delegation and other controls. This is independent of the way the communications are protected; therefore, it is usable with different key distribution methods.
- Security information needed to establish the association. The form of this depends on the key distribution method used. It is a Kerberos ticket if this is secret key based; it is a profile of the **SPKM_REQ** token for public key mechanisms. In both cases, there is a link between this and the PAC. Changing the security mechanism mainly just requires replacing this part of the token.
- Dialogue key packages to establish confidentiality and integrity keys.

3.4 SPKM Protocol

This section specifies the SPKM protocol, a simple public-key GSS-API mechanism. It is based on SPKM as defined in [20]. SPKM protocol provides CSI level 0 functionality only and the purpose is to allow the adoption of a simple security infrastructure without undue complexity or overhead.

SPKM has two separate GSS-API mechanisms, **SPKM_1** and **SPKM_2**, whose primary difference is that **SPKM_2** requires the presence of secure timestamps for the purpose of replay detection during context establishment and **SPKM_1** does not. **SPKM_1** is the mandatory mechanism for conformance to the SPKM protocol while **SPKM_2** is the optional mechanism.

Specifically, it defines the required information for encoding a secure interoperability **IOR** and defines the token formats used by the SECIOP protocol.

3.4.1 Cryptographic Profiles

The following cryptographic profiles are supported with this mechanism:

3.4.1.1 MD5_RSA

Specifies use of the SPKM mechanism to provide data integrity and authenticity by computing an **RSA** signature on the **MD5** hash of that data. The default SPKM key establishment algorithm is used (i.e., the context key is generated by the initiator, encrypted with the **RSA** public key of the target, and sent to the target). Note that **MD5_RSA** is a mandatory integrity and authenticity algorithm for SPKM.

3.4.1.2 MD5_DES_CBC

Specifies use of the SPKM mechanism to provide data integrity by encrypting, using **DES** in **CBC** mode, the **MD5** hash of that data. The default SPKM key establishment algorithm is used.

3.4.1.3 DES_CBC

Specifies use of the SPKM mechanism to provide data confidentiality by using **DES** in **CBC** mode. The default key establishment algorithm is used.

3.4.1.4 MD5_DES_CBC_SOURCE

Specifies use of the SPKM mechanism to provide data integrity by encrypting, using **DES** in **CBC** mode, the **MD5** hash of that data. The default key establishment algorithm is used plus source authentication information is also encrypted with the target's public key.

3.4.1.5 DES_CBC_SOURCE

Specifies use of SPKM mechanism to provide data confidentiality by using **DES** in **CBC** mode. The default key establishment algorithm is used plus source authentication information is also encrypted with the target's public key.

Values for these cryptographic profiles are assigned in Appendix B, Section B.2, “General Security Data Module,” on page B-1.

3.4.2 IOR Encoding

The security tags in the **IOR** are encoded. The component data member associated with the **SPKM_1** and **SPKM_2** mechanism tags is a **struct**, defined as follows:

```
struct <mechanism_name> {
    AssociationOptions                  target_supports;
    AssociationOptions                  target_requires;
    sequence <CryptographicProfile> crypto_profiles;
    sequence<octet>                     security_name;
};
```

mechanism_name can be either **SPKM_1** or **SPKM_2** and **security_name** must contain a valid **X.500** distinguished name represented as a string conforming to [4]. For example, it could be **“cn=Andrew Rust, ou=Home Office, o=Acme Widgets Inc., c=CA"**.

All tag components must be encapsulated using CDR encoding.

3.4.3 Using SPKM for SECIOP

When the SPKM protocol is chosen as the security mechanism for invoking an object, the SECIOP protocol carries the information described in this section. This protocol is a profile of the SPKM GSS-API mechanism as defined in [20].

All SPKM tokens are encoded according to the general format described in Section 3.3.7, “Mapping of Common Elements to the SECIOP Protocol,” on page 3-58.

The **innerContextTokens** are described in the following sections. All **innerContextTokens** are encoded using ASN.1 BER (constrained, in the interests of parsing simplicity, to the DER subset defined in [22]).

The SPKM GSS-API mechanism is identified by an OBJECT IDENTIFIER representing "**SPKM_1**" or “**SPKM_2**”. **SPKM_1** uses random numbers for replay detection during context establishment and **SPKM_2** uses timestamps (note that for both mechanisms, sequence numbers are used to provide replay and out-of-sequence detection during the context, if this has been requested by the application). **SPKM_1** OBJECT IDENTIFIER is **1.3.6.1.5.5.1.1** and **SPKM_2** OBJECT IDENTIFIER is **1.3.6.1.5.5.1.2**.

3.4.3.1 The Initial Context Token

The **initial_context_token** carried within an **EstablishContext** SECIOP message is encoded according to the general framework and confirms to the **SPKM-REQ** token as described in [20] Section 3.1.1.

In the **initial_context_token**, channel bindings are required to be ZERO (**GSS_C_NO_BINDINGS**).

The **GSS_C_DELEG_FLAG** is required to be FALSE (no delegation is supported).

The **GSS_C_MUTUAL_FLAG** is TRUE if it requires both parties to authenticate itself and **FALSE** (the default) if only one party is required to authenticate itself.

3.4.3.2 The Final Context Token

The **final_context_token** carried within a **CompleteEstablishContext** SECIOP message is encoded according to the **SPKM-REP-TI** token as defined in [20] Section 3.1.2 or the **SPKM-ERROR** token as defined in [20] Section 3.1.4.

3.4.3.3 The Continuation Context Token

The **continuation_context_token** carried within a **ContinueEstablishContext** SECIOP message is encoded according to the **SPKM-REP-TI** token or the **SPKM-REP-IT** token as defined in [20] Section 3.1.3 or the **SPKM-ERROR** token.

3.4.3.4 The Message Protection Token

The **message_protection_token** carried within a SECIOP **MessageInContext** message is encoded according to the **SPKM-MIC** token (for integrity) or **SPKM-WRAP** token (for confidentiality) as defined in [20] Section 3.2.

3.4.3.5 The Context Delete Token

The **context_delete_token** carried within a SECIOP **DiscardContext** message is encoded according to the **SPKM-DEL** token as defined in [20] Section 3.2.3.

3.5 GSS Kerberos Protocol

This section specifies the GSS Kerberos protocol. It is based on the GSS Kerberos specification [12] which itself is based on Kerberos V5 as defined in [13]. This specification refers to, rather than repeats, information in [12] and [13].

This section defines the required information for encoding the mechanism specific information in the IOR and the token formats used by the SECIOP protocol.

3.5.1 Cryptographic Profiles

The following cryptographic profiles are supported with this mechanism:

3.5.1.1 DES_CBC_DES_MAC

Specifies use of the Kerberos V5 mechanism with **DES MAC** message digest for integrity and **DES** in **CBC** mode for confidentiality.

3.5.1.2 DES_CBC_MD5

Specifies use of the Kerberos V5 mechanism with **MD5** message digest for integrity and **DES** in **CBC** mode for confidentiality.

3.5.1.3 DES_MAC

Specifies use of the Kerberos V5 mechanism with **DES MAC** message digest for integrity.

3.5.1.4 MD5

Specifies use of the Kerberos V5 mechanism with a **DES** encrypted **MD5** message digest for integrity.

Values for these cryptographic profiles are assigned in Appendix B, Section B.2, “General Security Data Module,” on page B-1.

3.5.2 Mandatory and Optional Cryptographic Profiles

ORB implementations claiming conformance to the GSS Kerberos protocol must implement at least the **MD5** profile. Conformant ORBs may, but are not required to, implement the remaining cryptographic profiles defined in this specification.

3.5.3 IOR Encoding

The security tags in the **IOR** are encoded. Both security name and association options tags may appear in the **IOR** and be shared between mechanisms.

The component data member associated with the **KerberosV5** mechanism tag is a **struct** defined as follows:

```
struct KerberosV5 {
    AssociationOptions                  target_supports;
    AssociationOptions                  target_requires;
    sequence<CryptographicProfile>  crypto_profiles;
    sequence<octet>                     security_name;
};
```

security_name shall contain a valid Kerberos Principal Name of type **GSS_KRBV5_NT_PRINCIPAL_NAME**, which is defined in [12].

All tag components must be encapsulated using CDR encoding.

3.5.4 SECIOP Tokens

When the GSS-Kerberos protocol is chosen as the security mechanism for invoking an object, the SECIOP protocol carries the information described in this section. All Kerberos tokens are encoded according to the general format.

The OBJECT IDENTIFIER for Kerberos V5 is **1.3.5.1.2** until [12] is advanced to a Proposed Standard RFC when it will be changed to **1.2.840.113554.1.2.2**.

Each individual token is distinguished by the data carried in the ANY field of this general framework.

3.5.4.1 The Initial Context Token

The **initial_context_token** carried within an **EstablishContext** SECIOP message is encoded according to the general framework and conforms to the unencrypted authenticator message as described in [12] Section 1.1.1.

Note that channel bindings are required to be ZERO (**GSS_C_NO_BINDINGS**) in this specification (see Section 3.3.8, "CSI Protocols," on page 3-60).

The **GSS_C_DELEG_FLAG** is set when either the client has called **set_security_features** specifying **SecDelModeSimpleDelegation** or when an administrator has called **set_delegation_mode** with a value of **SecDelModeSimpleDelegation** on a domain to which the target object belongs. The optional "Deleg" field, if present, includes a forwardable Ticket Granting Ticket (TGT) representing the delegated credentials of the client sending the **EstablishContext** message.

The **GSS_C_MUTUAL_FLAG** is set when either the client has called **set_association_options** specifying a value of **EstablishTrustInTarget** or an administrator has called **set_association_options** with a value of **EstablishTrustInTarget** on the domain to which the target belongs.

The **GSS_C_REPLAY_FLAG** and **GSS_C_SEQUENCE_FLAG** are generally clear as they can cause incorrect replay and misordering detection in a multi-threaded environment (see Section 3.3.3, "Association Options," on page 3-55).

Note – The current GSS Kerberos implementation available without cost from MIT does not support replay detection.

3.5.4.2 The Final Context Token

The **final_context_token** carried within a **CompleteEstablishContext** SECIOP message is encoded according to the formats defined in [12] Section 1.1.2.

3.5.4.3 The Continuation Context Token

Kerberos V5 does not use the **ContinueEstablishContext** message and therefore does not define the **continuation_context_token** format. If the Kerberos V5 mechanism is amended in the future to support mechanism negotiation, support of the **ContinueEstablishContext** message would be necessary and thus definition of the **continuation_context_token** would be required.

3.5.4.4 The Message Protection Token

The **message_protection_token** carried within a SECIOP **MessageInContext** message is encoded according to the formats defined in [12] section 1.2.

3.6 CSI-ECMA Protocol

This section defines the CSI-ECMA protocol. It is based on the ECMA GSS-API mechanism as defined in ECMA-235, though is a significant subset of that. It supports all CSI levels (0, 1, and 2). It provides three options for key distribution:

1. A secret key option using Kerberos data structures.
2. A hybrid option where secret keys are used within an administrative domain, but public keys are used between domains.
3. A public key option which uses public key technology for key distribution both within and between domains.

This section includes the full definition of the CSI-ECMA protocol so that it can be read without reference to ECMA 235. The CSI-ECMA protocol is a subset of ECMA 235. It is very similar to the SESAME profile as described in [16].

The CSI-ECMA protocol supports the CORBA Security Level 2 facilities. It is designed to be extensible as new facilities (for example, new delegation options) are agreed in future, and further key distribution options. It is also designed to respond to the requirements of international deployment such as minimal confidentiality (only keying information needs to be encrypted), use of anonymous audit (a separate **audit_id** can be transmitted), and choice of cryptography for message protection (including strong integrity, weak confidentiality).

The structure of the initial context token is key to providing this flexibility. It is separated into three parts:

1. Authorization information.
2. Information concerned with establishing the security association using one of the supported key distribution options.
3. Information concerned with generating the dialogue keys for message protection.

3.6.1 Concepts

3.6.1.1 Separation of Concerns

The initial context token transmitted in the SECIOP EstablishContext message on setting up a security association contains a number of parts with limited links between them. This is so that the different parts can be varied independently of each other. The three main parts are:

1. Authorization information - the Privilege Attribute Certificate (PAC) which contains the privileges used for access control and other attributes such as the audit id. Associated with this are delegation and other controls. Therefore, this is concerned with the access control and delegation policies, but is mainly independent of the key establishment and message protection mechanisms. The PAC can be updated to affect these policies independently of mechanisms. (The size of the PAC may be

significant; therefore, it is not confidentiality protected, as this may cause regulatory problems.) Privilege and other attributes in PACs are described in Section 3.6.2, “Security Attributes,” on page 3-68.

2. Target key block - used to provide the information needed to establish the security association between client and target. Secret key or public key technology (or some hybrid of these) may be used. The result is always a “basic” key from which dialogue keys to protect application messages can be derived. Therefore, this is concerned with the mechanism for establishing trust and distributing keys. This can be varied independently of the authorization policies and the message protection methods. Key establishment methods are described in Section 3.6.5, “Key Distribution Schemes,” on page 3-69.
3. Dialogue key packages which control how dialogue keys to protect messages are derived from the basic key. Note that this is largely independent of the key distribution method (i.e., public key technology may be used to establish secret keys for dialogue protection).

3.6.2 Security Attributes

3.6.2.1 Privilege Attributes

The CSI-ECMA protocol allows a range of privilege attributes in a Privilege Attribute Certificate (PAC) transmitted between the client and target object. These privileges then can be used for access control.

Privilege attributes which can be carried in the PAC at level 2 are defined in Appendix B, Section B.11.1, “Attribute Types,” on page B-27 and include all those defined in the CORBA Security specification.

A vendor or user enterprise may also define its own privilege attributes (if the particular implementation allows this) and use them for access control.

In line with the CORBA Security specification, each privilege attribute has a defining authority which identifies the authority responsible for defining the semantics of the value of the security attribute. This can be included for each privilege attribute in the PAC and in this case, there could be a different defining authority for each privilege.

It is often the case that most attributes in the PAC come under the same defining authority which is the authority that issued the PAC. If the PAC, as transmitted, does not have defining authorities for some attributes, then the issuing authority of the PAC is considered to be the defining authority.

3.6.2.2 Miscellaneous Attributes

This specification allows other types of security attributes to be carried in the PAC under the general heading of miscellaneous attributes. In CSI-ECMA, the only type of miscellaneous attribute supported is the audit identity.

3.6.3 Target Access Enforcement Function

The security processing functionality at the target is split between the target application and the target access enforcement function (**targetAEF**). ISO (ISO/IEC 10181-3) defines an access enforcement function collocated with the target application which controls access to a target application. This has a number of advantages including:

- The security critical code is isolated which makes security evaluation simpler.
- Long term keys can be shared between applications/objects. This can simplify administration (as there are less keys) and allow re-use of keying information when accessing another application/object sharing this **targetAEF**.

The **targetAEF** is responsible for setting up the security association, including validating the PAC and releasing the keys for message protection.

3.6.4 Basic and Dialogue Keys

The exchanges between client and target are secured using a two level key scheme in which a distinction is made between basic and dialogue keys.

A basic key is a temporary key established between a client and the target (actually, the **targetAEF**). The basic key is used for integrity protection of the PAC and associated information, its own key establishment information, and the information used to establish the dialogue keys. The basic key is established by the client sending information to the target in the **targetKeyBlock**. This can take different forms, depending on the key distribution method used.

A dialogue key is a temporary key established between the client and target and is used to protect the requests and responses. Separate dialogue keys can be established for integrity and confidentiality protection, enabling different strengths of mechanism to be configured. The information required to derive the dialogue keys is transmitted in the Dialogue key package. Typically, dialogue keys are constructed from the basic key using a one way algorithm.

3.6.5 Key Distribution Schemes

The CSI-ECMA protocol allows a choice of key distribution methods for establishing a client-target security association including the basic key. The content of the **targetKeyBlock** depends on the scheme used.

The key distribution schemes depend on the existence of long term cryptographic keys. Both secret (symmetric) and public (asymmetric) key technology can be used. When secret keys are used, a key is shared between the target and its Key Distribution Service (KDS). When public keys are used, the private key is kept by the principal and the public key held in a certificate, in a directory or elsewhere.

Initiators may also possess symmetric or asymmetric keys established as the result of an earlier authentication.

This CSI-ECMA specification defines three key distribution schemes. These are described below and are identified by a name and an architectural option number. Other schemes are possible as extensions to this as described in ECMA-235.

3.6.5.1 Basic Symmetric Key Distribution Scheme

In this scheme, the client and target each share different secret keys with the same Key Distribution Server. The scheme name for this is: symmIntradomain. The architectural option number is **2**.

To establish the association between the client and target, the client obtains a **targetKeyBlock** from its KDS containing a basic key encrypted under the target's long term key. On receipt of the targetKeyBlock, the target can extract the basic key from it.

In this case, the targetKeyBlock is a Kerberos ticket.

3.6.5.2 Symmetric Key Distribution with Asymmetric KDS

In this scheme, the initiator shares a secret key with its KDS and the target shares a secret key with its KDS (which is different). In addition, each KDS possesses a private/public key pair. The scheme name for this is: **hybridInterdomain**. The architectural option number is **3**.

To establish the client-target association, the client gets a **targetKeyBlock** from its KDS containing the basic key encrypted under a temporary key and the temporary key encrypted under the target's KDS public key. The **targetKeyBlock** is also signed using the initiator's KDS private key.

On receipt of the **targetKeyBlock**, the target transmits it to its KDS and gets back the basic key encrypted under the long term secret key it shares with its KDS.

3.6.5.3 Full Public Key Scheme

In this scheme, both client and target possess private/public keys. Neither use a KDS. The scheme name for this is: **asymmetric**. The architectural option number is **6**.

To establish the client-target association, the client constructs a **targetKeyBlock** containing a basic key encrypted under the target's public key. The target key block is signed with the client's private key. On receipt of the **targetKeyBlock**, the target directly establishes a basic key from it.

3.6.6 Cryptographic Algorithms and Profiles

Cryptographic and hashing algorithms are used for various purposes. This section categorizes the algorithms according to usage so that client and targets can determine more easily if they have the cryptographic support required to allow interoperation. The categorization then is refined into cryptographic profiles that can be incorporated into specific mechanism identifiers. The mechanism identifiers with cryptographic profiles then can be carried in the **IOR**. Table 3-6 summarizes the different uses to which algorithms are put.

Table 3-6 Summary of Algorithm Usage

Use Reference	Description of Use	Type of Algorithm
2	**PAC** protection using signature	OWF + asymmetric signature
3	basic key usage	confidentiality and integrity
4	integrity dialogue key derivation	OWF
5	integrity dialogue key usage	symmetric integrity
6	CA public keys	OWF + asymmetric signature
7	encryption using shared long term symmetric key	symmetric confidentiality
8	name hash to prevent ciphertext stealing	OWF
9	asymmetric basic key distribution	asymmetric encryption
10	key establishment within **SPKM_REQ**	(fixed value)
11	confidentiality dialogue key derivation	OWF
12	confidentiality dialogue key use	symmetric confidentiality

The algorithms can now be further categorized into broader classes, as shown in the following table.

Table 3-7 Summary of Algorithm Classes

Class 1:	symmetric for security of mechanism:	uses 3, 5, 7
Class 2:	all OWFs:	uses 2, 4, 6, 8, 11
Class 3:	internal mechanism asymmetric, encrypting:	use 9
Class 4:	internal mechanism asymmetric, non encrypting:	use 2
Class 5:	CA’s asymmetric non-encrypting:	use 6
Class 6:	data confidentiality, symmetric:	use 12

Use 10 is a fixed value and does not contribute to mechanism use options.

Based on these classes, the following cryptographic algorithm usage profiles are defined. Other profiles are possible and can be defined as required. Note that symmetric algorithm key sizes are included in this profiling, thus DES/64 indicates DES with a 64 bit key.

Table 3-8 Cryptographic Algorithm Usage Profiles

	Profile 1 Full	**Profile 2 no data confidentiality**	**Profile 3 low grade confidentiality**	**Profile 5 defaulted**
Class 1	DES/64	DES/64	RC4/128	separately agreed default
Class 2	MD5	MD5	MD5	separately agreed default
Class 3	RSA	RSA	RSA	separately agreed default
Classes 4 and 5	RSA	RSA	RSA	separately agreed default
Class 6	DES/64	None	RC4/40	separately agreed default

Table key:

- Profile 1 provides full security, using standard cryptographic algorithms with common accepted key sizes.
- Profile 2 is the same, but without supporting any confidentiality of user data.
- Profile 3 provides low grade confidentiality. In some countries, products using this are exportable without restriction; in others, they are more easily exportable/importable.
- Profile 5 uses algorithms identified by a separately specified default. It is intended for use by organizations who wish to use their own proprietary or government algorithms by separate agreement or negotiation.

3.6.7 PAC Protection and Delegation - Outline

The ECMA protocol provides a number of ways to protect a principal’s credentials, as held in a PAC. In CSI-ECMA, a digital signature is used, as this allows a target system to check what Security Authority authorized use of these privileges, without relying on the transitive trust needed for sealed PACs crossing domain boundaries. Encrypted PACs are not included in this profile.

There may also be controls on where the PAC may be delegated and used.

Protection method fields in the PAC specify where this PAC can be used and whether it can be used by the specified targets only (for example, allowing use of the privileges for access control) or whether that target can also delegate it.

Protection method fields are grouped together into method groups. The protection method check is passed if all the method fields in any one of the method groups is passed.

3.6.8 PPID Method

This method protects the PAC from being stolen, by restricting the initiators who can use the PAC.

When no other method group is present, it permits the PAC to be used only by the client entity to which it was originally issued (i.e., it prevents delegation). However, a PAC with a PPID will be delegatable if delegation is permitted by a PV/CV method.

A PPID identifying the initiating principal is put in the PAC by the Privilege Attribute (or other security) Service, according to policy or client request. The same/related information is also supplied as part of the **targetKeyBlock** so that the target can check that the entity which sent this token is the same entity which is entitled to use the PAC.

The PPID is a security attribute whose value in the CSI-ECMA protocol can take one of two forms, depending on the key distribution scheme used by the initiator.

- When the initiator has a secret key, the PPID is a random bit string which is also sent in the authorization field of the Kerberos ticket. This ticket is sent as part of the targetKeyBlock and can be checked to come from this client.
- For the public key scheme, the PPID contains the certificate serial number and CA name for the initiator's **X.509** public key certificate. The **targetKeyBlock** sent to the target is signed using this initiator's private key.

3.6.9 PV/CV Delegation Method

This method prevents the PAC from being stolen and at the same time controls whether (and where) it can be delegated. The method field in the PAC contains a protection value (PV) which is a one way function of a Control Value (CV).

A PAC will be accepted by the target (subject to other controls in the PV's method group) if the client proves knowledge of the CV by passing it (encrypted) as part of the initial context token. A method group contains at most one PV value.

In the simplest case, the method group contains just the PV and the target can delegate the PAC if it receives the CV.

The PV/CV method can be used for more selective targeting of the PAC also. A method group can include qualifier attributes which specify where the PAC can be used. **Qualifier** attributes can specify which principals can receive the PAC as a target and which can act as both delegate and target. These principals can be specified by their identities (though the protocol is extensible for other options such as a group/domain to which they belong).

For the simpler case, delegation can be prevented by setting the delegation mode to **Security::SecDelModeNoDelegation**. This will cause the client to send the PAC without the CV.

Note – The protocol allows more than one method group in the PAC, each with its own PV/CV. This can be used by a client or intermediate object in a chain to further restrict who can use the PAC, by failing to send some of the CVs. However, this specification does not include any operations for restricting delegation in this way, so it is not possible to exploit this capability.

3.6.9.1 Restrictions

Other restrictions may be included in the PAC. An ORB conforming to this specification does not have to generate these restrictions, but will reject PACs with mandatory restrictions which it does not understand or cannot process.

3.6.10 Mechanism Identifiers and IOR Encoding

All tag component data in the IOR must be encapsulated using CDR encoding.

Mechanism identifiers for the CSI-ECMA protocol have up to three parts, as follows:

1. The **protocol identifier**. This is CSI-ECMA.
2. The **architectural option**. This identifies the architectural option (i.e., the key distribution method used when establishing security associations). If absent, the default option is used.
3. The **cryptographic profile**. This identifies the cryptographic profile as defined above. If absent, a default is used.

In the IOR, the mechanism name in the struct of the **TAG_x_SEC_MECH** is:

CSI-ECMA_<architectural option>

where the architectural options supported are Secret, Hybrid, and Public; therefore, mechanism names are **CSI_ECMA_Secret**, **CSI_ECMA_Hybrid**, and **CSI_ECMA_Public**.

These values could also be negotiated using a generic mechanism negotiation scheme such as that in [19] in future, but are in the IOR for the current CSI specification.

3.6.11 Security Names

This protocol uses two forms of security names:

1. Directory names (DNs) are used where public key technology is used, as this is the form of name used in X509 certificates.

2. Kerberos names are used where secret key technology is used, as this is the form of name used by Kerberos.

3.6.11.1 Kerberos Naming

An entity that uses the normal Kerberos V5 authentication is given a printable Kerberos principal name of the form:

<principal_name>@realm_name>

Note – Components of a name can be separated by "/".

The separator @ signifies that the remainder of the string following the @ is to be interpreted as a realm identifier. If no @ is encountered, the name is interpreted in the context of the local realm. Once an @ is encountered, a non-null realm name, with no embedded "/" separators must follow. The "/" character is used to quote the character that follows immediately.

3.6.11.2 Directory Naming

Where public key technology supported by Directory Certificates is used, entities are given DNs. Such names are normally transmitted as directoryNames. At interfaces, they are strings built from components separated by a semicolon. The standardized keywords supported are:

CN	**(common-name)**
S	**(surname)**
OU	**(organization unit)**
O	**(organization)**
C	**(country)**

An example of a supported DN is:

CN=Martin;OU=Sesame;O=Bull;C=fr

There is no general rule for mapping the Directory name of an entity onto its Kerberos principal name. An explicit mapping is provided in a principal's Directory Certificate using the extensions field of the extended Directory Certificate syntax (version 3) to carry the principal's Kerberos name.

The syntax of the login name is imported from the Kerberos V5 GSS-API mechanism. The form of name is referred to using the symbolic name: GSS_KRB5_NT_PRINCIPAL. Syntax details are given in [12].

3.6.12 SECIOP Tokens When Using CSI-ECMA

All SECIOP security tokens conform to the basic token format defined in "Basic Token Format" on page 3-58. The object identifier for the **MechType** is of the form:

{generic_CSI_ECMA_mech (y) (z)}

where the value for **generic_CSI_ECMA_mech** is **1.3.12.0.235.4** and the values of y and z, if present, represent the architectural option number and cryptographic profile numbers. Both **y** and **z** can be defaulted.

The **innerContextToken** of the SECIOP message may be any of the tokens defined in Section 3.3.7.2, "Inner Context Tokens," on page 3-59. For context establishment, tokens are:

InitialContextToken	Sent by the initiator to a target, to start the process of establishing a Security Association.
TargetResultToken	Sent to the initiator by the target, if needed, following receipt of an Initial Context Token.
ErrorToken	Sent by the target on detection of an error during Security Association establishment.

The per-message tokens are:

MICToken	Sent either by the initiator or the target to verify the integrity of the user data sent separately.
WrapToken	Sent either by the initiator or the target. Encapsulates the input user data (optionally encrypted) along with integrity check values.

A **ContextDeleteToken** may also be used either by the initiator or the target to release a Security Association.

This definition uses ASN.1 types from other standards (e.g., the ISO definition of a Certificate). These types are detailed in Annex E of ECMA-235.

3.6.13 Initial Context Token

The initial context token contains:

- General information such as the token id, **contextFlags** (**delegation**, **replay-detect** etc.), **utcTime**, **seq-number**, etc.
- A **targetAEF** part to be passed to the target access enforcement function. This includes the PAC and associated CVs, target key block, and dialogue key package.
- A seal.

token id. etc.	**target AEF part** (used by target to enforce policy)			**seal**
	pac & CVs (initiating and/or delegate principal's authorization and delegation information)	**target Key Block** (information needed to establish the association)	**dialogue Key Block** (information used to establish message protection key - integrity and confidentiality)	

Figure 3-6 Initial Context Token

```
InitialContextToken ::=  SEQUENCE{
        ictContents         [0]   ICTContents,
        ictSeal             [1]   Seal
}
```

ictContents
Body of the initial context token

ictSeal
Seal of **ictContents** computed with the integrity dialogue key. Only the **sealValue** field of the **Seal** data structure is present. The cryptographic algorithms that apply are specified by **integDKUseInfo** in the **dialogueKeyBlock** field of the initial context token.

```
ICTContents ::= SEQUENCE {
        tokenId                 [0]   INTEGER, -- shall contain X'0100'
        SAId                    [1]   OCTET STRING,
        targetAEFPart           [2]   TargetAEFPart,
        targetAEFPartSeal       [3]   Seal,
        contextFlags            [4]   BIT STRING {
                                              delegation (0),
                                              mutual-auth (1),
                                              replay-detect (2),
                                              sequence (3),
                                              conf-avail (4),
                                              integ-avail (5)
                                       }
        utcTime                 [5]    UTCTime           OPTIONAL,
        usec                    [6]    INTEGER           OPTIONAL,
        seq-number              [7]    INTEGER           OPTIONAL,
        initiatorAddress        [8]    HostAddress       OPTIONAL,
        targetAddress           [9]    HostAddress       OPTIONAL
}
```

tokenId
Identifies the initial-context token. Its value is **01 00 (hex)**

SAId
A random number for identifying the Security Association being formed; it is one which (with high probability) has not been used previously. This random number is generated by the initiator and processed by the target as follows:

- If no **targetResultToken** is expected, the SAId value is taken to be the identifier of the Security Association being established (if this is unacceptable to the target, then an error token with etContents value of **gss_ses_s_sg_sa_already_established** must be generated).
- If a **targetResultToken** is expected, the target generates its random number and concatenates it to the end on the initiator's random number. The concatenated value is then taken to be the identifier of the Security Association being established.

targetAEFPart
Part of the initial-context token to be passed to the target access enforcement function. This is defined below and includes PAC, basic, and dialogue key packages.

targetAEFPartSeal
Seal of the **targetAEFPart** computed with the basic key. Only the **sealValue** field of the **Seal** data structure is present. The cryptographic algorithms that apply are specified by algorithm profile in the mechanism option.

contextFlags
Combination of flags that indicates context-level functions requested by the initiator.

Flag	Indicates that ...
delegation	when set to **0**, the initiator explicitly forbids delegation of the PAC in the **targetAEFPart**.
mutual-auth	mutual authentication is requested.
replay-detect	replay detection features are requested to be applied to messages transferred on the established Security Association.
sequence	sequencing features are requested to be enforced to messages transferred on the established Security Association.
conf-avail	a confidentiality service is available on the initiator side for the established Security Association.
integ-avail	an integrity service is available on the initiator side for the established Security Association.

utcTime
The initiator's UTC time.

usec
Micro second part of the initiator's time stamp. This field along with utcTime are used together to specify a reasonably accurate time stamp.

seq-number
When present, specifies the initiator's initial sequence number; otherwise, the default value of **0** is to be used as an initial sequence number.

initiatorAddress
Initiator's network address part of the channel bindings. This field is present only when channel bindings are transmitted by the caller to the mechanism implementation. Conformant ORBs do not need to generate this field.

targetAddress
Target's network address part of the channel bindings. This field is present only when channel bindings are transmitted by the caller to the implementation.

3.6.13.1 *TargetAEF Part*

```
TargetAEFPart ::= SEQUENCE {
        pacAndCVs                [0]   SEQUENCE OF CertandECV OPTIONAL,
        targetKeyBlock           [1]   TargetKeyBlock,
        dialogueKeyBlock         [2]   DialogueKeyBlock,
        targetIdentity           [3]   Identifier,
        flags                    [4]   BIT STRING {
                                                delegation        (0)
                                          }
}
```

pacAndCVs
The initiator ACI to be used for this Security Association. This field is not present when the association does not require any ACI. This field contains the PAC together with associated PAC protection information. When only simple delegation is supported, exactly one of these should be present.

If composite delegation options are supported, this field will contain more than one PAC. For example, for the initiator plus immediate invoker case, the initiator’s PAC would be present (with CVs) and the immediate invoker’s (with a PPID).

targetKeyBlock
The **targetKeyBlock** carrying the basic key to be used for the Security Association being established.

dialogueKeyBlock
A dialogue key block used by the **targetAEF** along with the basic key to establish an integrity dialogue key and a confidentiality dialogue key for per-message protection over the Security Association being established.

targetIdentity
The identity of the intended target of the Security Association. Used by the **targetAEF** to validate the PAC. Can also be used by the **targetAEF** to help protect the delivery of dialogue keys.

flags
Flags required by the **targetAEF** for its validation process. Contains only a delegation flag, the value of which is the same as the value of delegation flag in **contextFlag** field of **ictContents**. When the flag is set, all ECVs sent in **pacAndCVs** are made available to the target. Other bits are reserved for future use.

3.6.14 TargetResultToken

This token is returned by the target if the mutual-req flag is set in the Initial Context Token. It serves to authenticate the target to the initiator since only the genuine target could derive the integrity dialogue key needed to seal the **TargetResultToken**.

```
TargetResultToken ::=  SEQUENCE{
        trtContents             [0]    TRTContents,
        trtSeal                 [1]    Seal
}

TRTContents ::= SEQUENCE {
        tokenId                 [0]    INTEGER,    -- shall contain X'0200'
        SAId                    [1]    OCTET STRING,
        utcTime                 [5]    UTCTime        OPTIONAL,
        usec                    [6]    INTEGER        OPTIONAL,
        seq-number              [7]    INTEGER        OPTIONAL,
}
```

Note – There is no field for returning certification data here. This is because any such data that may be required is assumed to be returned at the conclusion of mechanism negotiation.

trtContents
This contains only administrative fields, identifying the token type, the context, and providing exchange integrity.

seq-number
When present, specifies the target's initial sequence number; otherwise, the default value of **0** is to be used as an initial sequence number.

The other administrative fields are as described previously.

trtSeal
Seal of **trtContents** computed with the integrity dialogue key. Only the **sealValue** field of the **Seal** data structure is present. The cryptographic algorithms that apply are specified by **integDKUseInfo** in the **dialogueKeyBlock** field of the initial context token.

3.6.15 ErrorToken

An error token may be returned, as follows:

```
ErrorToken ::=   {
        tokenType               [0]    OCTET STRING VALUE X'0400',
        etContents              [1]    ErrorArgument,
}
```

etContents
Contains the reason for the creation of the error token. The different reasons are given as minor status return values.

```
ErrorArgument ::= ENUMERATED {
        gss_ses_s_sg_server_sec_assoc_open                   (1),
        gss_ses_s_sg_incomp_cert_syntax                      (2),
        gss_ses_s_sg_bad_cert_attributes                     (3),
        gss_ses_s_sg_inval_time_for_attrib                   (4),
        gss_ses_s_sg_pac_restrictions_prob                   (5),
        gss_ses_s_sg_issuer_problem                          (6),
        gss_ses_s_sg_cert_time_too_early                     (7),
        gss_ses_s_sg_cert_time_expired                       (8),
        gss_ses_s_sg_invalid_cert_prot                       (9),
        gss_ses_s_sg_revoked_cer                             (10),
        gss_ses_s_sg_key_constr_not_supp                     (11),
        gss_ses_s_sg_init_kd_server_ unknown                 (12).
        gss_ses_s_sg_init_unknown                            (13),
        gss_ses_s_sg_alg_problem_in_dialogue_key_block       (14),
        gss_ses_s_sg_no_basic_key_for_dialogue_key_block     (15),
        gss_ses_s_sg_key_distrib_prob                        (16),
        gss_ses_s_sg_invalid_user_cert_in_key_block          (17),
        gss_ses_s_sg_unspecified                             (18),
        gss_ses_s_g_unavail_qop                              (19),
        gss_ses_s_sg_invalid_token_format                    (20)
}
```

3.6.16 Per Message Tokens

The syntax of the **message_protection_token** in SECIOP messages has the same general structure for both MIC and Wrap tokens:

```
PMToken ::=  SEQUENCE{
        pmtContents             [0]    PMTContents,
        pmtSeal                 [1]    Seal
                -- seal over the pmtContents being protected
}

PMTContents ::= SEQUENCE {
        tokenId                 [0]    INTEGER, -- shall contain X'0101'
        SAId                    [1]    OCTET STRING,
        seq-number              [2]    INTEGER         OPTIONAL
        userData                [3]    CHOICE {
                                               plaintext        BIT STRING,
                                               ciphertext       OCTET STRING  OPTIONAL
                                       }
        directionIndicator      [4]    BOOLEAN        OPTIONAL
}
```

pmtContents

tokenId

SAId

A random number for identifying the Security Association being formed; it is one which (with high probability) has not been used previously. This random number is generated by the initiator and processed by the target as follows:

- If no **targetResultToken** is expected, the **SAId** value is taken to be the identifier of the Security Association being established (if this is unacceptable to the target, then an error token with **etContents** value of **gss_ses_s_sg_sa_already_established** must be generated).
- If a **targetResultToken** is expected, the target generates its random number and concatenates it to the end on the initiator's random number. The concatenated value is then taken to be the identifier of the Security Association being established.

seq-number

This field must be present if replay detection or message sequencing have been specified as being required at Security Association initiation time. The field contains a message sequence number whose value is incremented by one for each message in a given direction, as specified by **directionIndicator**. The first message sent by the initiator following the **InitialContextToken** shall have the message sequence number specified in that token, or if this is missing, the value **0**. The first message returned by the target shall have the message sequence number specified in the **TargetReplyToken** if present, or failing this, the value **0**.

The receiver of the token will verify the sequence number field by comparing the sequence number with the expected sequence number and the direction indicator with the expected direction indicator. If the sequence number in the token is higher than the expected number, then the expected sequence number is adjusted and **GSS_S_GAP_TOKEN** is returned. If the token sequence number is lower than the expected number, then the expected sequence number is not adjusted and **GSS_S_DUPLICATE_TOKEN** or **GSS_S_OLD_TOKEN** is returned, whichever is appropriate. If the direction indicator is wrong, then the expected sequence number is not adjusted and **GSS_S_UNSEQ_TOKEN** is returned.

userData

See specific token type narratives below.

directionIndicator

FALSE indicates that the sender is the context initiator, **TRUE** that the sender is the target.

pmtSeal

See specific token type narratives below.

3.6.16.1 MICToken

A **MICToken** is a per-message token, separate from the user data being protected, which can be used to verify the integrity of that data as received. The token is passed in the **message_protection_token** in SECIOP messages, and the protected data follows as a GIOP message or message fragment. The syntax of the token is:

```
MICToken ::= PMToken
```

The overall structure and field contents of the token are described above. Fields specific to the **MICToken** are:

userData
Not present for **MICTokens**.

pmtSeal
The **Checksum** is calculated over the **DER** encoding of the **pmtContents** field with the user data temporarily placed in the **userData** field. The **userData** field is not transmitted.

3.6.16.2 WrapToken

A **WrapToken** encapsulates the input user data (optionally encrypted) along with associated integrity check values. It consists of an integrity header followed by a body portion that contains either the plaintext or encrypted data. The syntax of the token is:

```
WrapToken ::= PMToken
```

The overall structure and field contents of the token are described above. Fields specific to the **WrapToken** are:

userData
Present either in plain text form or encrypted. If the data is encrypted, it is performed using the Confidentiality Dialogue Key, and as in [13], an 8-byte random confounder is first prepended to the data to compensate for the fact that an **IV** of **zero** is used for encryption.

wtSeal
The **Checksum** is calculated over the **pmtContents** field, including the **userData**. If the **userData** field is to be encrypted, the seal value is computed prior to the encryption.

3.6.17 ContextDeleteToken

The **ContextDeleteToken** is issued by either the context initiator or the target to indicate to the other party that the context is to be deleted.

```
ContextDeleteToken ::=  SEQUENCE {
        cdtContents      [0]   CDTContents,
        cdtSeal          [1]    Seal
                     --  seal over cdtContents, encrypted under the Integrity
                     --  Dialogue Key. Contains only the sealValue field
}

CDTContents ::= SEQUENCE {
        tokenType        [0]    OCTET STRING VALUE X'0301',
        SAId             [1]   OCTET STRING,
        utcTime          [2]   UTCTime       OPTIONAL,
        usec             [3]   INTEGER       OPTIONAL,
        seq-number       [4]    INTEGER         OPTIONAL,
}
```

cdtContents
This contains only administrative fields, identifying the token type, the context, and providing exchange integrity.

seq-number
When present, this field contains a value one greater than that of the **seq-number** field of the last token issued from this issuer. The other administrative fields are as described above.

trtSeal
See above for a general description of the use of this construct.

3.6.18 Security Attributes

3.6.18.1 Data Structures

The security attribute is a basic construct for privilege and other attributes in PACs.

```
SecurityAttribute ::= SEQUENCE{
        attributeType                   Identifier,
        attributeValue                  SET OF SEQUENCE {
                                        definingAuthority  [0] Identifier   OPTIONAL,
                                        securityValue                  [1] SecurityValue
        }
}

Identifier ::= CHOICE{
        objectId                        [0]    OBJECT IDENTIFIER,
        directoryName                   [1]    Name,
                        -- imported from the Directory Standard
        printableName                   [2]    PrintableString,
        octets                          [3]    OCTET STRING,
        intVal                          [4]    INTEGER,
        bits                            [5]    BIT STRING,
        pairedName                      [6]    SEQUENCE{
                                                printableName       [0] PrintableString,
                                                uniqueName          [1] OCTET STRING
                                               }
}

SecurityValue ::= CHOICE{
        directoryName                   [0]    Name,
        printableName                   [1]    PrintableString,
        octets                          [2]    OCTET STRING,
        intVal                          [3]    INTEGER,
        bits                            [4]    BIT STRING,
        any                             [5]    ANY -- defined by attributeType
}
```

Only one set member is permitted in AttributeValue. Multivalue attributes are effected in the **securityValue** field, where the "SEQUENCE OF" construct can be used. (Including "SET OF" in the syntax enables security attributes to be stored as normal in a Directory whenever the choice made within Identifier is OBJECT IDENTIFIER.)

A directory name is translated into a string format as defined in Section 3.6.11, "Security Names," on page 3-74. The **sequence<octet>** attribute value returned at the IDL interface is a representation of this string, not the more complex ASN.1 definition of this.

attributeType
Defines the type of the attribute. Attributes of the same type have the same semantics when used in Access Decision Functions, though they may have different defining authorities.

definingAuthority
The authority responsible for the definition of the semantics of the value of the security attribute. This optional field of the **attributeValue** can be used to resolve potential value clashes. It is defined as an Identifier which has a choice of syntax. For CSI-ECMA, it is always a **directoryName**.

securityValue
The value of the security attribute. Its syntax can be either one of the basic syntaxes for attributes or a more complex one determined by the attribute type.

3.6.18.2 Attribute Types

An attribute type in this standard is formally defined as an Identifier which provides a choice of syntax; however, all standard attribute types are defined as OBJECT IDENTIFIERs. Three types of attributes are defined:

1. Privilege attributes (e.g., **AccessId**, **GroupId**, **Role**)
2. Miscellaneous attributes, mainly the **AuditId**
3. Qualifier attributes used within the **PV**/**CV** delegation scheme to say where credentials can be used/delegated.

For standard attributes, the OBJECT IDENTIFIER includes

- first, a standard part with the value **1.3.12.1.46**,
- then the "family" for privilege, miscellaneous, or qualifier attributes (**4**, **3**, or **5**), and
- then the value for that particular attribute type.

All standard attributes, which conformant ORBs must be able to generate/transmit, have this form.

In addition, conformant ORBs must be able to handle other attribute types defined in this chapter. They must also be able to handle attribute types with "**OMG**" object identifiers, as described in Section 3.1.13.5, "Mapping Other Attributes to Externally Valid IDL Attributes," on page 3-29. In this case, the Object Identifier is:

<iso>..<omg>.<security><family definer>.<family>.<attribute type>

where the values of the CORBA family definer, CORBA family and attribute type are as defined in Appendix B, Section B.11.1, "Attribute Types," on page B-27. For standard attributes, the family definer is **0** and the family is **0** for privileges and **1** for miscellaneous attributes.

OMG Object Identifiers can also be used for privilege attributes defined by other organizations, who have registered a family definer with OMG.

3.6.19 Privilege and Miscellaneous Attribute Definitions

Privilege and miscellaneous attribute types are normally identified by Object Identifiers which have a standard part, then family and attribute type parts.

The following privilege and miscellaneous attributes are defined in the CORBA Security specification and have defined attribute types. Some of these are mandatory for a CSI level 2 conformant ORB to generate (see Section 3.1.15, "Support for CORBA Security Facilities and Extensibility," on page 3-32). The Object Identifier in the privilege attribute set for that type is listed in the following table.

Table 3-9 Privilege and Miscellaneous Attributes

Type of Attribute	oid family & type	Syntax	Meaning
access-identity	4.2	printableString	The access identity represents the principal's identity to be used for access control purposes.
primary-group	4.3	printableString	The primary group represents a unique group to which a principal belongs. A security context must not contain more than one primary group for a given principal.
group	4.4	SEQUENCE OF printableString	A group represents a characteristic common to several principals. A PAC may contain more than one group for this principal.
role	4.1	printableString	A role attribute represents one of the principal's organizational responsibilities.
audit_id	3.2	printableString	The identity of the principal as used for auditing.

3.6.20 Qualifier Attributes

When a **targetQualification** or **delegateTargetQualification** method is present in the PAC, the syntax used for the method parameters is **securityAttribute**. Object Identifiers for qualifier attributes have the value **1.3.12.1.46.5**.<qualifier attribute type>.

Currently, only one form of qualifier attribute is defined, and this identifies the target by security name. This is usually the name of an identity domain as defined in Section 2.1.8, "Domains," on page 2-21, not an individual object.

In future, other forms of qualifier attributes may be added. For example, the attribute could identify an invocation delegation domain, rather than particular named target.

3.6.21 Target Names

Within a PAC protection method, a target name is indicated using the OID:

```
target-name-qualifier OBJECT IDENTIFIER ::= {qualifier-attribute 1 }
Its syntax in the PAC is:
TargetNameValueSyntax ::= Identifier
```

3.6.22 PAC Format

The PAC is in the form of a generalized certificate. A Generalized Certificate is composed of three main structural components:

1. The “**commonContents**” fields collectively serve to provide generally required management and control over the use of the PAC.
2. The “**specificContents**” fields are different for different types of certificate, and contain a type identifier to indicate the type. In this specification, only one type is defined - the Privilege Attribute Certificate (PAC).
3. The “**checkValue**” fields are used to guarantee the origin of the certificate. This is a signature in the CSI-ECMA specification. (though a seal would be possible as in ECMA 235).

Common Certificate Contents	PAC specific contents			Check Value
	protection/ delegation methods	privilege and other attributes	restrictions	

Figure 3-7 Generalized Certificate’s Structural Components

```
GeneralizedCertificate ::= SEQUENCE{
        certificateBody         [0]   CertificateBody,
        checkValue              [1]   CheckValue
}

CertificateBody ::= CHOICE{
        encryptedBody           [0]   BIT STRING,
        normalBody              [1]   SEQUENCE{
                                      commonContents      [0] CommonContents,
                                      specificContents    [1] SpecificContents
                                }
}
```

The next sections describe these three main structural components of the Generalized Certificate.

3.6.23 Common Contents fields

```
CommonContents ::= SEQUENCE{
        comConSyntaxVersion     [0]   INTEGER { version1 (1) }DEFAULT 1,
        issuerDomain            [1]   Identifier           OPTIONAL,
        issuerIdentity          [2]   Identifier,
        serialNumber            [3]   INTEGER,
        creationTime            [4]   UTCTime              OPTIONAL,
        validity                [5]   Validity,
        algId                   [6]   AlgorithmIdentifier,
        hashAlgId               [7]   AlgorithmIdentifier       OPTIONAL
}
```

In the imported definition of **AlgorithmIdentifier**, ISO currently permits both a hash and a cryptographic algorithm to be specified. If this is done, they must appear in the algId field. The **hashAlgId** field is present for those cases where a separate hash algorithm specification is required.

```
Validity ::= SEQUENCE {
        notBefore         UTCTime,
        notAfter          UTCTime
} -- as in [ISO/IEC 9594-8]
 -- Note: Validity is not tagged, for compatibility with the
-- Directory Standard.
```

comConFieldsSyntaxVersion
Identifies the version of the syntax of the combination of the **commonContents** and the **checkValue** fields parts of the certificate.

issuerDomain
The security domain of the issuing authority. Not required if the form of issuerIdentity is a full distinguished name, but required if other forms of naming are in use. In CSI-ECMA, this is always a directoryName.

issuerIdentity
The identity of the issuing authority for the certificate.

serialNumber
The serial number of the certificate (**PAC**) as allocated by the issuing authority.

creationTime
The **UTCtime** that the certificate was created, according to the authority that created it.

validity
A pair of start and end times within which the certificate is deemed to be valid.

algId
The identifier of the secret or of the public cryptographic algorithm used to seal or to sign the certificate. If there is a single identifier for both the encryption algorithm and the hash function, it appears in this field.

hashAlgId
The identifier of the hash algorithm used in the seal or in the signature.

The certificate can be uniquely identified by a combination of the **issuerDomain**, **issuerIdentity**, and **serialNumber**.

3.6.24 Specific Certificate Contents for PACs

```
SpecificContents ::= CHOICE{
        pac                     [1]    PACSpecificContents
        -- only the PAC is used here
}

PACSpecificContents ::= SEQUENCE{
        pacSyntaxVersion        [0]    INTEGER{  version1 (1)}         DEFAULT 1,
        protectionMethods       [2]    SEQUENCE OF MethodGroup         OPTIONAL,
        pacType                 [4]    ENUMERATED{
                                       primaryPrincipal  (1),
                                       temperedSecPrincipal  (2),
                                       untemperedSecPrincipal(3)
                                }       DEFAULT 3,
        privileges              [5]    SEQUENCE OF PrivilegeAttribute,
        restrictions            [6]    SEQUENCE OF Restriction         OPTIONAL,
        miscellaneousAtts       [7]    SEQUENCE OF SecurityAttribute   OPTIONAL,
        timePeriods             [8]    TimePeriods                     OPTIONAL
}

PrivilegeAttribute ::= SecurityAttribute

Restriction ::= SEQUENCE {
        howDefined      [0] CHOICE {
                                included          [3] BIT STRING
                        },
                                -- the actual restriction in a form undefined here
        type            [2] ENUMERATED  {
                                mandatory (1),
                                optional    (2)
                        }       DEFAULT mandatory,
        targets         [3] SEQUENCE OF SecurityAttribute    OPTIONAL
}               -- applies to all targets if this is omitted
```

pacSyntaxVersion
Syntax version of the PAC.

protectionMethods
A sequence of optional groups of **Method** fields used to protect the certificate from being stolen or misused. For a full description see below.

pacType
Indicates whether the privileges contained in the PAC are those of a Primary Principal (e.g., the client) or of a Secondary Principal (e.g., the user). In this specification, it is always a PAC of a secondary principal untempered by the privileges of a Primary Principal.

privileges
Privilege Attributes of the principal.

restrictions
This field enables the original owner of the PAC to impose constraints on the operations for which it is valid. There are two types of restriction:

- Mandatory: If a target to which the restriction applies cannot understand the bit string defining the restriction, access should not be granted.
- Optional: If a target application to which the restriction applies cannot understand the bit string, it is expected to ignore it.

For CSI-ECMA, it is not mandatory to generate restrictions, but mandatory restrictions cannot be ignored. If not understood, the PAC cannot be accepted.

miscellaneousAtts
Security attributes which are neither privileges attributes nor restrictions attributes. In a PAC, this may include identity attributes such as Audit Identity. For the CSI-ECMA specification, this is the only miscellaneous attribute expected.

timePeriods
This field adds further time restrictions to the validity field of the **commonContents**. Either **startTime** or **endTime** can be optional. The **TimePeriods** control is passed if the time now is within any of the sequence periods, or if there is a period with a start before now and no **endTime**, or there is a period with an end after now and no **startTime**.

3.6.24.1 Protection Methods

A method consists of a method id and parameters (**methodParams**). The method id determines the syntax for the type of **methodParams**.

```
Method ::= SEQUENCE{
        methodId        [0]   MethodId,
        methodParams    [1]   SEQUENCE OF Mparm     OPTIONAL
}
MethodId ::= CHOICE{
        predefinedMethod      [0] ENUMERATED {
                                      controlProtectionValues    (1),
                                      ppQualification            (2),
                                      targetQualification        (3),
                                      delegateTargetQualification  (4)
                              }
}

Mparm ::= CHOICE{
        pValue                [0]   PValue,
        securityAttribute     [1]   SecurityAttribute
}


PValue ::= SEQUENCE{
        pv                    [0]   BIT STRING
        algorithmIdentifier   [1]   AlgorithmIdentifier    OPTIONAL
}
```

```
CertandECV ::=   SEQUENCE {
        certificate                [0]   GeneralizedCertificate,
        ecv                        [1]   ECV        OPTIONAL
}
        - ECV is defined in later
```

methodId
Identifies a protection method. Methods can be used in any combination, and except where stated otherwise, multiple occurrences of the same method are permitted. The choice of methodId determines the permitted choices of method parameters in the methodParams construct as described below.

methodParams
Parameters for a protection method. The semantics of each protection method is described in section Section 3.1.9.2, "Cryptographic Profiles," on page 3-15.

For the Primary Principal Qualification Method, the **MethodId** is **ppQualification** and the syntax of **Mparm** is **securityAttribute**. Its value is defined in Section 3.6.8, "PPID Method," on page 3-73.

For the PV/CV method, the **MethodId** is:**controlProtectionValues** and the syntax of Mparm is: **pValue**.

For the Target Qualification protection method, the **MethodId** is **targetQualification** and the syntax for **Mparms** is **securityAttribute**.

For the Delegate/Target Qualification protection method, the **MethodId** is delegate**targetQualification** and the syntax for **Mparms** is **securityAttribute**.

The security attribute in the target and delegate/target protection method is a qualifier attribute as defined in Section 3.6.20, "Qualifier Attributes," on page 3-86.

3.6.24.2 External Control Values Construct

When using the **controlProtectionValues** method a PAC protected under that method may be accompanied by one or more control values and indices to the method occurrences in the certificate to which they apply. Also, when such a certificate is being issued to a requesting client, the CV values it will need in order to use that certificate may need to be returned with it.

```
ECV ::= SEQUENCE {
        crypAlgIdentifier          [0]   AlgorithmIdentifier   OPTIONAL,
        cValues                    [1]   CHOICE {
                                         encryptedCvalueList        [0] BIT STRING,
                                         individualCvalues          [1] CValues
                                   }
}

CValues ::= SEQUENCE OF SEQUENCE {
        index                      [0]   INTEGER,
        value                      [1]   BIT STRING
}
```

crypAlgIdentifier
This specifies the encryption algorithm of the control values.

cValues
An ECV construct can contain either an encrypted list of control values in the **encryptedCvalueList** field, or a list of individual control values in **individualCvalues**.

If the **encryptedCvalueList** choice is made, the whole list is encrypted in bulk, but the in-clear contents of this field are expected to have the syntax **CValues**. If the **individualCvalues** choice is made, values are individually encrypted in the value fields of the list. Encryption is always done under the basic key protecting the operation.

In the case of the **controlProtectionValues** method, value is a CV, and index is then the index of the method occurrence in the certificate, starting at 1.

3.6.25 Check Value

In this specification, a PAC is protected by being digitally signed by the issuer.

A signature may be accompanied by information identifying the Certification Authority under which the signature can be verified, and with an optional convenient reference to or the actual value of the user certificate for the private key that the signing authority used to sign the certificate.

```
CheckValue ::= CHOICE{
        signature                       [0]   Signature
        -- only signature supported here
}

Signature ::= SEQUENCE{
        signatureValue                  [0]   BIT STRING,
        publicAlgId                     [1]   AlgorithmIdentifier   OPTIONAL,
        hashAlgId                       [2]   AlgorithmIdentifier   OPTIONAL,
        issuerCAName                    [3]   Identifier   OPTIONAL,
        caCertInformation               [4]   CHOICE {
                                              caCertSerialNumber    [0]   INTEGER,
                                              certificationPath     [1]   CertificationPath
                                        } OPTIONAL
}
--CertificationPath is imported from [22]
```

signatureValue
The value of the signature. It is the result of a public encryption of a hash value of the **certificateBody**.

publicAlgId
Only present if the certificate body is encrypted, then it is a duplication of the **algId** value in "**commonContents**." This is not required in CSI-ECMA.

hashAlgId
Only present if the certificate body is encrypted, then it is a duplication of the **hashAlgId** value in “**commonContents**.” This is not required in CSI-ECMA.

issuerCAName
The identity of the Certification Authority that has signed the user certificate corresponding to the private key used to sign this certificate.

caCertInformation
Contains either just a certificate serial number which together with the **issuerCAName** uniquely identifies the user certificate corresponding to the private key used to sign this certificate, or a full specification of a certification path via which the validity of the signature can be verified. The latter option follows the approach used in [22].

The **Seal** structure is used in the **Tokens** defined above.

```
Seal ::= SEQUENCE{
        sealValue                [0]   BIT STRING,
        secretAlgId              [1]   AlgorithmIdentifier      OPTIONAL,
        hashAlgId                [2]   AlgorithmIdentifier      OPTIONAL,
        targetName               [3]   Identifier               OPTIONAL,
        keyId                    [4]   INTEGER                  OPTIONAL
}
```

sealValue
The value of the seal. It is the result of a secret encryption of a hash value of a set of octets (which are the **DER** encoding of some ASN.1 type)

secretAlgId
An optional indicator of the sealing algorithm.

hashAlgId
Only present if the **secretAlgId** does not specify which hashing algorithm is used.

targetName
This field identifies the **targetAEF** or target with which the secret key used for the seal is shared.

keyId
This serial number together with the **targetName** uniquely identifies the secret key used in the seal.

3.6.26 Basic Key Distribution

The **TargetKeyBlock** is structured as follows:

- An identifier (**kdSchemeOID**) for the key distribution scheme being used, which takes the form of an OBJECT IDENTIFIER.
- A part which, if present, the target AEF needs to pass on to its KDS (**targetKDSPart** - will be present only when the target AEF's KDS is different from the initiator's).
- A part which, if present, can be used directly by the **targetAEF** (**targetPart**).

When a **targetAEF** using a separate **KDS** receives the **targetKeyBlock**, it first checks whether it supports the key distribution scheme indicated in **kdsSchemeOID**. Two different cases need to be considered:

1. Only the **targetPart** is present. The target **AEF** computes the basic key directly, using the information present in the **targetPart**. The syntax of **targetPart** is scheme dependent. Expiry information optionally can be present in **targetPart**. If supported by the scheme, the Primary Principal attributes of the initiator will also be present for **PAC** protection under the Primary Principal Qualification method (see above).
2. Only the **targetKDSPart** is present. The **targetAEF** forwards the **TargetKeyBlock** to its KDS. In return, it receives a scheme dependent data structure which allows the target AEF to determine the basic key and, if supported by the scheme, the Primary Principal attributes of the initiator for PAC protection purposes. Expiry information can optionally be present in the **targetKDSPart**.

The form of this information depends on the key distribution configuration in place.

3.6.27 Keying Information Syntax

```
TargetKeyBlock ::= SEQUENCE {
    kdSchemeOID      [2]  OBJECT IDENTIFIER,
    targetKDSpart    [3]  ANY        OPTIONAL,
                                -- depending on kdSchemeOID
    targetPart       [4]  ANY        OPTIONAL
                                -- depending on kdSchemeOID
}
```

kdSchemeOID
Identifies the key distribution scheme used. Allows the **targetAEF** to determine rapidly whether or not the scheme is supported. It also allows for the easy addition of future schemes.

targetKDSpart
Part of the Target Key Block which is processable only by the **KDS** of the target **AEF**. This part is sent by the target **AEF** to its local **KDS**, in order to get the basic key which is in it. It must always contain the name of a target “served” by the **targetAEF** in question. The mapping between the name of the application and the name of the target AEF is known to the target **AEF**'s **KDS** which is able to authenticate which **targetAEF** is issuing the request for translating the **targetKDSpart**. It can then verify that the AEF is one which is responsible for the application name contained in the **targetKDSpart**. If it is, the key is released and is sent protected back to the requesting **AEF**. **TargetKDSpart** should include data that enables the **KDS** of the target **AEF** to authenticate the **KDS** of the initiator. When the “Primary Principal Qualification” protection method needs to be used for the **PAC**, unless there is an accompanying **targetPart**, **targetKDSpart** must contain the appropriate primary principal security attributes (which is always true in this specification).

targetPart
A part of the Target Key Block which is processed only by the target **AEF**. When there is no **targetKDSpart** it is processable directly; otherwise, it can only be processed after the target **KDSpart** has been processed by the **KDS** of the target **AEF**, and the appropriate Keying Information has been returned to the **AEF**. The targetPart construct

should include data that enables the target AEF to authenticate the **KDS** of the initiator. When the "Primary Principal Qualification" protection method needs to be used for the PAC, **targetPart** must contain the primary principal security attributes.

3.6.28 Summary of Key Distribution Schemes

This specification defines three key distribution schemes. These are:

1. **symmIntradomain**: using a secret key technology within a domain. In this case, the **targetKDSpart** of the **TargetKeyBlock** is not supplied and the **targetPart** contains a Kerberos ticket.
2. **hybridInterdomain**: In this case, the **targetPart** field is not supplied. The **PublicTicket** contains a Kerberos ticket.
3. asymmetric: the **targetKDSpart** is not supplied and the **targetPart** contains an **SPKM_REQ**.

The following table shows the different syntaxes used for **targetKDSpart** and **targetPart** for the defined KD-schemes. "Missing" in the table means that the relevant construct is not supplied.

Table 3-10 Syntaxes Used for targetKDSpart and targetPart

KD-Scheme name	**kdSchemeOID**	**targetKDSpart**	**targetPart**
symmIntradomain	{kd-schemes 1}	Missing	Ticket
hybridInterdomain	{kd-schemes 3}	PublicTicket	Missing
asymmetric	{kd-schemes 6}	Missing	SPKM_REQ

Further options are possible by defining further kd-schemes. For example, ECMA 235 also defines options for:

- initiators with public keys and targets with secret keys
- initiators with secret keys and targets with public keys

3.6.29 CSI-ECMA Secret Key Mechanism

In this scheme, the client and target each share different secret keys with the same Key Distribution Server.

To establish the association, between the client and target, the client obtains a **targetKeyBlock** from its KDS containing a basic key encrypted under the target's long term key. On receipt of the **targetKeyBlock**, the target can extract the basic key from it.

The *symmIntradomain* key distribution scheme

- has a mechanism id of **CSI_ECMA_Secret**, and

- uses a Kerberos ticket in the **targetKeyBlock** of the **initial_context_token**. An unmodified Kerberos **TGS** can be used as the **KDS** in this case.

3.6.29.1 Profile of Ticket as Used in SymmIntradomain Scheme

The following table indicates which optional fields must be present in the Kerberos ticket for the **CSI_ECMA_Secret** mechanism and indicates the values which are required to be present in all fields.

Table 3-11 Kerberos Ticket’s Mechanism Fields

Field	**Value/Constraint**
tkt-vno	5
realm	ticket issuer's domain name in Kerberos realm name form
sname	target application name including the realm of the target
- EncTicketPart	encrypted with long term key of target AEF
-- flags	only bits 6, 10 and 11 can be meaningful in the context of the CSI-ECMA protocol, the rest are ignored
-- key	the basic key
-- crealm	initiator domain name in Kerberos realm name form
-- cname	principal name of the initiator (in the case of delegation the cname will be that of the delegate)
-- transited	not used
-- authtime	the time at which the initiator was authenticated
-- starttime	not used
-- endtime	the time at which the ticket becomes invalid
-- renew-till	not used
-- caddr	not used
-- authorization-data	contains the PPID corresponding to cname

The Kerberos Ticket's **authorization_data** field contains the **PPID** of the context initiator, as formally defined below.

```
ECMA-AUTHORIZATION-DATA-TYPE ::= INTEGER { ECMA-ADATA (65) }
ECMA-AUTHORIZATION-DATA ::= SEQUENCE {
                ecma-ad-type        [0]   ENUMERATED  {ppidType  (0)},
                ecma-ad-value       [1]   CHOICE  {ppidValue [0]     SecurityAttribute
            }
}
```

ppidType
Indicates the type of the authorization data which is included in the **Ticket**.

ppidValue
This value is used in the **ppQualification PAC** protection method, as described above.

3.6.30 CSI-ECMA Hybrid Mechanism

In this scheme, the initiator shares a secret key with its **KDS** and the target shares a secret key with its **KDS** (which is different). In addition, each **KDS** possesses a private/public key pair.

To establish the client-target association, the client gets a **targetKeyBlock** from its **KDS** containing the basic key encrypted under a temporary key and the temporary key encrypted under the target's **KDS** public key. The **targetKeyBlock** is also signed using the initiator's **KDS** private key.

On receipt of the **targetKeyBlock**, the target transmits it to its **KDS** and gets back the basic key encrypted under the long term secret key it shares with its **KDS**.

The **hybridInterdomain** key distribution scheme

- has a mechanism id of **CSI_ECMA_Hybrid** in the **IOR**, and
- uses a Public ticket in the **targetKeyBlock** of the **initial_context_token**, as described below.

  A modified Kerberos **TGS** can be used as the **KDS** in this case.

3.6.30.1 Hybrid Inter-domain Key Distribution Scheme Data Elements

```
PublicTicket ::= SEQUENCE{
        krb5Ticke          [0]   Ticket,
        publicKeyBlock [1]   PublicKeyBlock
}

PublicKeyBlock ::= SEQUENCE{
        signedPKBPart                [0]    SignedPKBPart,
        signature                    [1]    Signature            OPTIONAL,
        certificate                  [2]    Certificate          OPTIONAL
}

SignedPKBPart ::= SEQUENCE{
        keyEstablishmentData         [0]    KeyEstablishmentData,
        encryptionMethod             [1]    AlgorithmIdentifier  OPTIONAL,
        issuingKDS                   [2]    Identifier,
        uniqueNumber                 [3]    UniqueNumber,


        validityTime                 [4]    TimePeriods,
        creationTime                 [5]    UTCTime
}

UniqueNumber ::= SEQUENCE{
        timeStamp                    [0]    UTCTime,
        random                       [1]    BIT STRING
}
```

krb5Ticket
The Kerberos Ticket which contains the basic key. The encrypted part of this ticket is encrypted using the key found within the **encryptedPlainKey** field of the **KeyEstablishmentData** in the **PublicKeyBlock**.

publicKeyBlock
Contains the key used to protect the **krb5Ticket** encrypted using the public key of the recipient and signed by the encryptor (i.e., the context initiator's KD-Server).

signedPKBPart
The part of the **publicKeyBlock** which is signed. The **keyEstablishmentData** field contains the **KeyEstablishmentData** (i.e., the actual encrypted temporary key).

- The **encryptionMethod** indicates the algorithm used to encrypt the **encryptedKey**.
- The **issuingKDS** is the name of the KD-Server which produced the **PublicTicket**.
- The **uniqueNumber** is a value (containing a timestamp and a random number) which prevents replay of the **PublicTicket**.
- **validityTime** specifies the times for which the **PublicTicket** is valid.
- creationTime contains the time at which the **PublicTicket** was created.

signature
Contains the signature calculated by the **issuingKDS** on the **signedPKBPart** field.

certificate
If present, contains the public key certificate of the issuing **KDS**.

3.6.30.2 Key Establishment Data Elements

These are used in public key establishment mechanisms.

```
KeyEstablishmentData ::= SEQUENCE {
        encryptedPlainKey     [0]   BIT STRING,-- encrypted PlainKey
        targetName            [1]   Identifier   OPTIONAL,
        nameHashingAlg        [2]   AlgorithmIdentifier       OPTIONAL
}

HashedNameInput ::= SEQUENCE {
        hniPlainKey           [0]  BIT STRING,-- same as plainKey
        hniIssuingKDS         [1]  Identifier


PlainKey ::= SEQUENCE {
        plainKey              [0]  BIT STRING,  -- The cleartext key
        hashedName            [1]  BIT STRING
}
```

encryptedPlainKey
Contains the encrypted key. The BIT STRING contains the result of encrypting a **PlainKey** structure.

targetName
If present, contains the name of the target application. This is necessary for some of the KD-schemes.

nameHashingAlg
Specifies the algorithm which is used to calculate the **hashedName** field of the **PlainKey**.

hniPlainKey
hniIssuingKDS
Used as input to a hashing algorithm as a general means to prevent ciphertext stealing attacks.

plainKey
Contains the actual bits of the plaintext key which is to be established.

hashedName
A hash of the name of the encrypting **KDS** calculated using the plainkey and **KDS** name as input (within the **HashedNameInput** structure). The algorithm identified in **nameHashingAlg** is used to calculate this value.

targetName
If present, contains the name of the target for which the **PublicTicket** was originally produced. This may be different from the targetIdentity field of the **initialContextToken** if caching of **PublicTickets** has been implemented.

3.6.30.3 Key Establishment Algorithm

The **PublicKeyBlock** in this mechanism and the **SPKM_REQ** construct used in scheme 6 requires a sequence of key establishment algorithm identifier values to be inserted into the **key_estb_set** field. The OBJECT IDENTIFIER below is defined as the (single) key establishment “algorithm” for ECMA mechanisms:

```
gss-key-estb-alg AlgorithmIdentifier ::= {kd-schemes, NULL }
```

gss-key-estb-alg
This AlgorithmIdentifier identifies the key establishment algorithm value to be used within the **key_estb_set** field of an **SPKM_REQ** data element as the one defined by ECMA.

This algorithm is used to establish a symmetric key for use by both the initiator and the target AEF as part of the context establishment. The corresponding **key_estb_req** field of the **SPKM_REQ** will be a BIT STRING the content of which is a **DER** encoding of the **KeyEstablishmentData** element.

3.6.30.4 Profile of Ticket as Used in Hybrid Interdomain Scheme

Note that the **krb5Ticket** part of this is identical to that used in the **CSI_ECMA_Secret** key mechanism except that the **EncTicketPart** is encrypted with the temporary key used between **KDS** rather than the target's key.

Table 3-12 Ticket as Used in Hybrid Interdomain Scheme

Field	Value/Constraint
krb5Ticket	
- tkt-vno	5
- realm	initiator domain name in Kerberos realm name form
- sname	target application name including the realm of the target
-- EncTicketPart	encrypted with temporary key (which is in turn encrypted within the keyEstablishmentData field)
--- flags	only bits 6, 10 and 11 can be meaningful in the context of the CSI-ECMA protocol, the rest are ignored
--- key	the basic key
--- crealm	initiator domain name in Kerberos realm name form
--- cname	principal name of the initiator (in the case of delegation the cname will be that of the delegate)
--- transited	not used
--- authtime	the time at which the initiator was authenticated
--- starttime	not used
--- endtime	the time at which the ticket becomes invalid
--- renew-till	not used
--- caddr	not used
--- authorization-data	contains the PPID corresponding to cname
publicKeyBlock	
- signedPKBPart	
-- encryptedKey	KeyEstablishmentData structure
-- encryptionMethod	gss-key-estb-alg
-- issuingKDS	X.500 name of initiator's KDS (the signer)
-- uniqueNumber	creation time of publicKeyBlock plus a random bit string
-- validityTime	only one period allowed

Table 3-12 Ticket as Used in Hybrid Interdomain Scheme *(Continued)*

Field	Value/Constraint
-- creationTime	creation time of publicKeyBlock
- signature	contains all the signing information as well as the actual signature bits
- certificate	optional

3.6.31 CSI-ECMA Public Mechanism

In this scheme, both client and target possess a private/public key pair and neither use a **KDS**.

To establish the client-target association, the client constructs a **targetKeyBlock** containing a basic key encrypted under the target's public key. The target key block is signed with the client's private key. On receipt of the **targetKeyBlock**, the target directly establishes a basic key from it.

The **asymmetric** key distribution scheme:

- has a mechanism id of **CSI_ECMA_Public**, and
- uses an **SPKM_REQ** in the **targetKeyBlock** of the **initial_context_token**.

This mechanism has only a profile of the **SPKM_REQ** as defined below.

3.6.31.1 Profile of SPKM_REQ Used in Public Key Mechanism

The following table indicates which optional fields must be present in the **SPKM_REQ** in the **targetKeyBlock** for the **CSI_ECMA_Public** mechanism and indicates the values which are required to be present in all fields.

Table 3-13 SPKM-REQ Used in Public Key Mechanism

Field	Value/Constraint
requestToken	
- tok_id	not used - fixed value of '0'
- context_id	not used - fixed value of bit string containing one zero bit
- pvno	not used - fixed value of bit string containing one zero bit
- timestamp	creation time of SPKM_REQ - required
- randSrc	random bit string
- targ_name	X.500 Name of target AEF
- src_name	X.500 Name of initiator
- req_data	
-- channelId	not used - octet string of length one value '00'H

Table 3-13 SPKM-REQ Used in Public Key Mechanism *(Continued)*

Field	**Value/Constraint**
-- seq_number	missing
-- options	not used - all bits set to zero
-- conf_alg	not used - use NULL CHOICE
-- intg_alg	not used - use a SEQUENCE OF with zero elements
- validity	mandatory
- key_estb_set	only one element supplied containing gss-key-estb-alg
- key_estb_req	contains KeyEstablishmentData with targetApplication field missing
- key_src_bind	missing
req_integrity	sig_integ mandatory
certif_data	only userCertificate field supported
auth_data	missing

Definitions of **KeyEstablishmentData** and **gss-key-estb-alg** are given in Section 3.6.30, "CSI-ECMA Hybrid Mechanism," on page 3-97.

3.6.32 Dialogue Key Block

Dialogue Key Block constructs are used to specify how the integrity dialogue key and confidentiality dialogue key should be derived from the basic key, and specify the cryptographic algorithms with which the keys should be used. Dialogue keys are explained above. The syntax is as follows:

```
 DialogueKeyBlock   ::=   SEQUENCE {
        integKeySeed                    [0]   SeedValue,
        confKeySeed                     [1]   SeedValue,
        integKeyDerivationInfo          [2]   KeyDerivationInfo       OPTIONAL,
        confKeyDerivationInfo           [3]   KeyDerivationInfo       OPTIONAL,
        integDKuseInfo                  [4]   DKuseInfo               OPTIONAL,
        confDKuseInfo                   [5]   DKuseInfo               OPTIONAL
}

SeedValue  ::= SEQUENCE {
        timeStamp                       [0]   UTCTime                 OPTIONAL,
        random                          [1]   BIT STRING
}

KeyDerivationInfo::= SEQUENCE {
        owfId                           [0]   AlgorithmIdentifier,
        keySize                         [1]   INTEGER
}

DKuseInfo    ::= SEQUENCE {
    useAlgId                    [0]  AlgorithmIdentifier,
    useHashAlgId                [1]  AlgorithmIdentifier      OPTIONAL
}
```

integKeySeed
A random number, optionally concatenated with a time value to ensure uniqueness, used as input to the one way function specified in **integKeyDerivationInfo**.

confKeySeed
A random number, optionally concatenated with a time value to ensure uniqueness, used as input to the one way function specified in **confKeyDerivationInfo**.

integKeyDerivationInfo
Key derivation information for the integrity dialogue key, as follows:

> **owfId**
> The one way algorithm which takes the basic key XOR the seed as input, resulting in the integrity dialogue key.
>
> **keySize**
> The size of the key in bits. If the algorithm identified by owfId produces a larger key, it is reduced by masking to this length, losing its most significant end.

confKeyDerivationInfo
Key derivation information for the confidentiality dialogue key. The fields in this construct have the same meanings as defined above for the integrity dialogue key.

integDKuseInfo
Information describing how the integrity dialogue key is to be used, as follows:

> **useAlgId**
> The secret or public reversible encryption algorithm with which the integrity dialogue key is to be used.
>
> **useHashAlgId**
> The one way function with which the integrity dialogue key is to be used. It is the hash produced by this algorithm on the data to be protected which is encrypted using **useAlgId**.

confDKuseInfo
Information describing how the confidentiality key is to be used. The **useHashAlgId** construct is not used here.

3.7 Integrating SSL with CORBA Security

3.7.1 Introduction

This section defines how SSL [21] is integrated with CORBA Security. SSL provides CSI level 0 (see Appendix D, Section D.7.2, “Common Secure Interoperability Levels,” on page D-12) functionality only. This level of functionality is achieved only if the optional authentication features of SSL are used.

3.7.2 Cryptographic Profiles

All of the cryptographic profiles defined by SSL may be used by ORBs using SSL for Security.

3.7.3 IOR Encoding

A new kind of security tag is defined, for use in the component tag sequence in the IIOP IOR profile body, to describe the use of Secure Transports with CORBA Security. This enables the future use of combinations of security mechanisms and secure transports.

The IIOP TAG identifying the SSL secure transport is **TAG_SSL_SEC_TRANS**. The tag component data described below must be encapsulated using CDR encoding. The data structure associated with this tag is as follows:

```
struct SSL {
    Security::AssociationOptions     target_supports;
    Security::AssociationOptions     target_requires;
    unsigned short                   port;
};
```

The definition of association options is the same as for the CSI protocols. SSL only supports client and target authentication if the optional certificate exchanger features of SSL are supported.

Unlike the CSI mechanism TAGs, the SSL TAG does not include cryptographic profiles as cryptography is negotiated as part of the SSL session establishment. For the same reason the TAG does not include a security name for the target.

The **port** field contains the port number to be used instead of the port defined in the accompanying IIOP profile body, if SSL is selected by the client. It contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.

As with the other secure interoperability options, if the client invokes the target without the appropriate level of security (e.g., if the client is not secure and simply invokes the target ignoring all security TAGs in the profile) the target shall raise the CORBA::NO_PERMISSION exception.

3.7.4 Relation to SECIOP

As SSL provides a secure transport layer over TCP/IP, the CORBA SECIOP protocol is not required when using SSL. Instead, the connection rules of IIOP (see the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*) are applied to SSL (which itself uses TCP).

3.8 DCE-CIOP with Security

This section describes how to provide secure interoperability between ORBs which use the DCE Common Inter-ORB Protocol (DCE-CIOP). It describes how the DCE-CIOP transport layer should handle security (for example, how it should interpret the security components of the **IOR** profile when selecting DCE Security Services for a request and secure invocation).

3.8.1 Goals of Secure DCE-CIOP

The original goals of DCE-CIOP, documented in the *Common Object Request Broker: Architecture and Specification*, are maintained and enhanced by Secure DCE-CIOP:

- Support multi-vendor, mission critical, enterprise-wide, secure ORB-based applications.
- Leverage services provided by DCE wherever appropriate.
- Allow efficient and straightforward implementation using public DCE APIs.
- Preserve ORB implementation freedom.

Secure DCE-CIOP achieves these goals by taking advantage of the integrated security services provided by DCE Authenticated RPC. It is not a goal of the Secure DCE-CIOP specification to support the use of arbitrary security mechanisms for protection of DCE-CIOP messages.

3.8.2 Secure DCE-CIOP Overview

Secure interoperability between ORBs using the DCE-CIOP transport relies on the DCE Security Services and the DCE Authenticated RPC runtime that utilizes those services.

The DCE Security Services (specified in [6]), as employed by the DCE Authenticated RPC runtime (specified in [7] and the [8]), provide the following security features:

- cryptographically secured mutual authentication of a client and target,
- ability to pass client identity and authorization credentials to the target as part of a request,
- protection against undetected, unauthorized modification of request data,
- cryptographic privacy of data, and
- protection against replay of requests and data.

The RPC runtime provides the communication conduit for exchanging security credentials between communicating parties. It protects its communications from threats such as message replay, message modification, and eavesdropping.

The DCE-CIOP uses DCE RPC APIs to request security features for a given client-target communication binding. Subsequent DCE-CIOP messages on that binding flow over RPC and thus are protected at the requested levels.

This Secure DCE-CIOP specification defines the **IOR** Profile components required to support Secure DCE-CIOP. Each component is identified by a unique tag, and the encoding and semantics of the associated **component_data** are specified. Client secure association requirements, as indicated by client-side policy, and target secure association requirements, as specified in the target **IOR** Profile security components, are mapped to DCE Security Services. Finally, the use of DCE APIs to protect DCE-CIOP messages is described.

3.8.2.1 IOR Security Components for DCE-CIOP

The information necessary to invoke secure operations on objects using DCE-CIOP is encoded in an **IOR** in a profile identified by **TAG_MULTIPLE_COMPONENTS**. The **profile_data** for this profile is a **CDR** encapsulation (see "CDR Transfer Syntax" in the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*) of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures. These types are described in the ORB Interoperability Architecture chapter of the *Common Object Request Broker: Architecture and Specification*.

The Multiple Component Profile contains the tagged components required to support DCE-CIOP, described in the DCE ESIOP chapter of the *Common Object Request Broker: Architecture and Specification*, as well as the components required to support security for DCE-CIOP. The general security components are described in "Security Components of the IOR" on page 3-8. The DCE-specific security component and semantics for the common security components are described here.

Although a conforming implementation of Secure DCE-CIOP is only required to generate and recognize the components defined here and in the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*, the profile may also contain components used by other kinds of ORB transports and services. Implementations should be prepared to encounter profiles identified by **TAG_MULTIPLE_COMPONENTS** that do not support DCE-CIOP. Unrecognized components should be preserved but ignored. Although an implementation may choose to order the components in a profile in a particular way, other implementations are not required to preserve that order. Implementations must be prepared to handle profiles whose components appear in any order.

TAG_DCE_SEC_MECH

For a profile to support Secure DCE-CIOP, it must include exactly one **TAG_DCE_SEC_MECH** component. Presence of this component indicates support for the (non-GSSAPI) "DCE Security with Kerberos V5 with DES" mechanism type. The **component_data** field contains an authorization service identifier and an optional sequence of tagged components.

Future versions of DCE Security that require different information than what is provided by the **component_data** structure described below are expected to be supported with a new component tag, rather than with revisions to the data structure associated with the **TAG_DCE_SEC_MECH** tag.

The DCE Security Mechanism component is defined by the following OMG IDL:

```
module DCE_CIOPSecurity {

    const IOP::ComponentId TAG_DCE_SEC_MECH = 103

    // CORBA IDL doesn't (yet) support const octet
    //
    // const octet DCEAuthorizationNone = 0;
    // const octet DCEAuthorizationName = 1;
    // const octet DCEAuthorizationDCE = 2;

    typedef unsigned short     DCEAuthorization;

    const DCEAuthorization     DCEAuthorizationNone = 0;
    const DCEAuthorization     DCEAuthorizationName = 1;
    const DCEAuthorization     DCEAuthorizationDCE = 2;

    // since consts of type octet are not allowed in IDL the constant
    // values that can be assigned to the authorization_service field
    // in the DCESecurityMechanismInfo is declared as unsigned shorts.
    // when they actually get assigned to the authorization_service field
    // they should be assigned as octets.

    struct DCESecurityMechanismInfo {
        octet                          authorization_service;
        sequence <TaggedComponent>     components;
    };
};
```

A **TaggedComponent** structure is built for the DCE Security Mechanism component by setting the tag member to **TAG_DCE_SEC_MECH**, and setting the **component_data** member to a **CDR** encapsulation of a **DCESecurityMechanismInfo** structure.

The authorization_service Field

The **authorization_service** field is used to indicate what authorization service is required by the target, and therefore must be supported by the authenticated RPC runtime for invocations on this **IOR**. Two authorization models are supported: **DCEAuthorizationName** and **DCEAuthorizationDCE** with a third identifier, **DCEAuthorizationNone**, to indicate that no authorization is required.

The components Field

The **components** field contains a sequence of zero or more tagged components, none of which may appear more than once, from the following list of common security **IOR** components: **TAG_ASSOCIATION_OPTIONS**, and **TAG_SEC_NAME**.

Each of these components, defined in Section 3.1.4.1, “Security Components of the IOR,” on page 3-8, may be present either in the components field of the **DCESecurityMechanismInfo** structure, or at the top level of the **IOR** profile. When one of these components appears at the top level of the profile, its data may be shared by other security mechanisms in the profile. When it appears in the nested components field of **DCESecurityMechanismInfo**, its data is available only to the DCE Security mechanism and overrides the data of an identically-tagged component, if present, at the top level of the profile.

3.8.2.2 TAG_ASSOCIATION_OPTIONS

The association options component, described in Section 3.1.4.1, “Security Components of the IOR,” on page 3-8, contains flags indicating which protection and authentication services the target supports and which it requires. This component is optional for Secure DCE-CIOP; defaults are used when the component is not present.

The way in which association options are interpreted for use with DCE security is reflected in Table 3-14 shows how an association option is mapped to a DCE RPC protection level and authentication service.

Table 3-14 Association Option Mapping to DCE Security

Association Option	DCE RPC Protection Level	DCE RPC Authentication Service
NoProtection	rpc_c_protect_level_none	rpc_c_authn_none
Integrity	rpc_c_protect_level_pkt_integrity	rpc_c_authn_dce_secret
Confidentiality	rpc_c_protect_level_pkt_privacy	rpc_c_authn_dce_secret
DetectReplay	rpc_c_protect_level_pkt	rpc_c_authn_dce_secret
DetectMisordering	rpc_c_protect_level_pkt	rpc_c_authn_dce_secret
EstablishTrustInTarget	rpc_c_protect_level_connect	rpc_c_authn_dce_secret
EstablishTrustInClient	rpc_c_protect_level_connect	rpc_c_authn_dce_secret
tag not present	rpc_c_protect_level_default	rpc_c_authn_dce_secret

If the **TAG_ASSOCIATION_OPTIONS** component is not present, then the target is assumed both to support and to require **rpc_c_protect_level_default** and **rpc_c_authn_dce_secret**. (The value of **rpc_c_protect_level_default** is defined by the DCE implementation or by a site administrator.)

The target_supports Field

When an association option is set in the **target_supports** field of the **TAG_ASSOCIATION_OPTIONS component_data**, it indicates that the target supports invocations which use Secure DCE-CIOP with the protection level and authentication service that correspond to the selected option, as shown in Table 3-14. Any or all of the association options may be set in the **target_supports** field. The options set in the **target_supports** field will be compared with client-side policy required options to determine if the target can support the client’s requirements.

Although, for the DCE security mechanism, a single selected option may imply support for several other options (e.g., selection of the Integrity option implies support for **DetectReplay**, **DetectMisordering**, and **EstablishTrustInClient**) it is recommended that every supported option be explicitly set in the **target_supports** field to facilitate comparison with client requirements.

The target_requires Field

When an association option is set in the **target_requires** field of the **TAG_ASSOCIATION_OPTIONS component_data**, it indicates that the target requires invocations secured with at least the protection level and authentication service that correspond to the selected option, as shown in Table 3-14. Since DCE RPC supports a range of protection levels, each of which provides all the protection of the level below it and also some additional protection, selecting multiple **target_requires** options does not make sense. For DCE, no more than one option need be selected in the **target_requires** field.

If a **TAG_ASSOCIATION_OPTIONS** component is contained within the **DCESecurityMechanismInfo** structure, the **target_requires** field may conform to the DCE semantics (i.e., no more than one option selected). If other security mechanisms are sharing the **TAG_ASSOCIATION_OPTIONS** component, and perhaps using different rules for interpreting the **target_requires** field, then the **target_requires** field may have several options selected. The “DCE Association Options Reduction” algorithm, described in Section 3.8.3.1, “Secure DCE-CIOP Operational Semantics,” on page 3-112, handles both cases and is used to select the appropriate DCE secure invocation services given a set of required association options.

The **EstablishTrustInTarget** option in the **target_requires** field is meaningless, and is therefore ignored.

3.8.2.3 TAG_SEC_NAME

The security name component contains the DCE principal name of the target. Generally, this is a global principal name that includes the name of the cell in which the target principal’s account resides. If a cell-relative principal name (i.e., the cell prefix does not appear) is specified, the local cell is assumed. Cell-relative principal names are only appropriate for use in **IOR**s that are consumed by clients in the same cell in which the target resides. When an **IOR** containing a cell-relative principal name in the **TAG_SEC_NAME** component crosses a cell boundary, the cell-relative principal name should be replaced with a global name.

The format of a “human-friendly” DCE principal name is described in section 1.13 of [6]. It is a string containing a concatenated cell name and cell-relative principal name that looks like:

/.../cell-name/cell-relative-principal-name

For example, the principal with the cell-relative name “**printserver**” in the “**mis.prettybank.com**” cell has the global principal name:

/.../mis.prettybank.com/printserver

The **component_data** member of the **TAG_SEC_NAME** component is set to the string value of the DCE principal name. The string is represented directly in the sequence of octets, including the terminating NULL.

If the **TAG_SEC_NAME** component is not present, then a value of NULL is assumed, indicating that the client will depend on the DCE authenticated RPC runtime to retrieve the DCE principal name of the target, identified in the IOR by the DCE-CIOP string binding and binding name components. This case indicates that the client is not interested in authentication of the target identity.

3.8.3 DCE RPC Security Services

This section provides details about the protection provided by DCE Authenticated RPC authorization services, protection levels, and authentication services. See the **rpc_binding_set_auth_info()** man page in [9] for more information about using these protection parameters to secure an association between a client and target.

DCE RPC Authorization Services

This section describes the DCE authorization service indicated by the **authorization_service** member of the **DCESecurityMechanismInfo** structure in the **component_data** field of the **TAG_DCE_SEC_MECH** component.

DCEAuthorizationName indicates that the target performs authorization based on the client security name. The DCE RPC authorization service **DCEAuthorizationName** asserts the principal name (without cryptographic protection if the association option **NoProtection** is chosen, or with cryptographic protection otherwise).

DCEAuthorizationDCE indicates that the target performs authorization using the client’s Privilege Attribute Certificate (for OSF DCE **1.0.3** or previous versions), or the client’s Extended Privilege Attribute Certificate (for DCE 1.1). The authorization service **DCEAuthorizationDCE** asserts the principal name and appropriate authorization data (without cryptographic protection if the association option **NoProtection** is chosen, or with cryptographic protection otherwise).

DCEAuthorizationNone indicates that the target performs no authorization based on privilege information carried by the RPC runtime. This is valid only if the association option **NoProtection** is chosen.

The authorization_service identifiers defined here for Secure DCE-CIOP correspond to DCE RPC authorization service identifiers and are defined to have identical values. The relationship between these identifiers is shown in the following table.

Table 3-15 Relation between DCE-CIOP and DCE RPC Authorization Service Identifiers

Secure DCE-CIOP authorization_service	**DCE RPC Authorization Service**	**Shared Value**
DCEAuthorizationNone	rpc_c_authz_none	0

Table 3-15 Relation between DCE-CIOP and DCE RPC Authorization Service Identifiers

Secure DCE-CIOP authorization_service	**DCE RPC Authorization Service**	**Shared Value**
DCEAuthorizationName	rpc_c_authz_name	1
DCEAuthorizationDCE	rpc_c_authz_dce	2

DCE RPC Protection Levels

The meanings of the DCE RPC protection levels referenced in Table 8-4 are described below. For the purposes of evaluating the protection levels, it is interesting to remember that a single DCE-CIOP message is transferred over the wire in the body of one or more DCE RPC PDUs.

rpc_c_protect_level_none indicates that no authentication or message protection is to be performed, regardless of the authentication service chosen. Depending on target policy, the client may be granted access as an unauthenticated principal.

rpc_c_protect_level_connect indicates that the client and server identities are exchanged and cryptographically verified at the time the binding is set up between them. Strong mutual authentication and replay detection *for the binding set-up only* is provided. There are no protection services per DCE RPC PDU.

rpc_c_protect_level_pkt indicates that the **rpc_c_protect_level_connect** services are provided plus detection of misordering or replay of DCE RPC PDUs. There is no protection against PDU modification.

rpc_c_protect_level_pkt_integrity offers the **rpc_c_protect_level_pkt** services plus detection of DCE RPC PDU modification.

rpc_c_protect_level_pkt_privacy offers the **rpc_c_protect_level_pkt_integrity** services plus privacy of RPC arguments, which means the DCE-CIOP message in its entirety is privacy protected.

rpc_c_protect_level_default indicates the default protection level, as defined by the DCE implementation or by a site administrator (should be one of the above defined values).

DCE RPC Authentication Services

The meanings of the DCE RPC authentication services referenced in Table 3-15 are described below.

rpc_c_authn_none indicates no authentication. If this is selected, then no authorization, DCEAuthorizationNone, must be chosen as well.

rpc_c_authn_dce_secret indicates the DCE shared-secret key authentication service.

3.8.3.1 Secure DCE-CIOP Operational Semantics

This section describes how the DCE-CIOP transport layer should provide security for invocation and locate requests.

During a request invocation, if the **IOR** components indicate support for the DCE-CIOP transport and the **TAG_DCE_SEC_MECH** component is present, then a Secure DCE-CIOP request can be made.

Deriving DCE Security Parameters from Association Options

The client-side secure invocation policy and the target-side policy expressed in the **TAG_ASSOCIATION_OPTIONS** component are used to derive the actual options using the method described in "Determining Association Options" on page 3-12. These options are then reduced to a single **required_option** using the algorithm described in "The DCE Association Options Reduction Algorithm" on page 3-112 below. The resultant **required_option** is used to select a DCE RPC protection level and authentication service using Table 3-14 on page 3-108. The derived protection level and authentication service are used to secure the association via the **rpc_binding_set_auth_info()** call (see "Securing the Binding Handle to the Target" on page 3-113).

The DCE Association Options Reduction Algorithm

The "DCE Association Options Reduction" algorithm is used to select a single association option, **required_option**, given the value required by client and target derived as described in "Determining Association Options" on page 3-12. The resultant **required_option** indicates, via Table 3-14 on page 3-108, the DCE protection level and authentication service to use for invocations.

The association option names used in the following algorithm refer to options in the negotiated-required options set.

The "DCE Association Options Reduction" algorithm is expressed as:

```
If Confidentiality is set, then required_option = Confidentiality;
else if Integrity is set, then required_option = Integrity;
else if DetectReplay is set, OR
    if DetectMisordering is set,
    then required_option = DetectReplay;
    (alternatively, the same results are obtained with:
    then required_option = DetectMisordering;)
else if EstablishTrustInClient is set,
    then required_option = EstablishTrustInClient;
else required_option = NoProtection.
```

Behavior When TAG_ASSOCIATION_OPTIONS Not Present

As described earlier, if the **TAG_ASSOCIATION_OPTIONS** component is not present, then the target is assumed to support and require **rpc_c_protect_level_default** and **rpc_c_authn_dce_secret**. Since these protection parameters are not expressed as association options, the usual method of deriving a single **required_option** by combining client and target policy (see “Determining Association Options” on page 3-12 and “The DCE Association Options Reduction Algorithm” on page 3-112“above) cannot be used. Instead, use the following alternative method to derive the required DCE RPC protection level and authentication service:

- Translate the client-side secure invocation policy from a set of client supported association options to a single **client_supported_option** and from a set of client required association options to a single **client_required_option**, using in each case the algorithm described in “The DCE Association Options Reduction Algorithm” on page 3-112.
- Using Table 3-14 on page 3-108 translate the **client_supported_option** and **client_required_option** to corresponding “supported” and “required” DCE RPC protection level/authentication service pairs.
- If the target principal is a member of the local cell, determine the target required protection level implied by **rpc_c_protect_level_default** by calling **rpc_mgmt_inq_dflt_protect_level()** passing **rpc_c_authn_dce_secret** as the **authn_svc** parameter. If the target principal is not a member of the local cell or if it’s difficult to determine, then assume a target required protection level of **rpc_c_protect_level_pkt_integrity**.
- If the client supports **rpc_c_authn_dce_secret**, then choose the strongest protection level that both the client and target support and that does not exceed the strongest protection level required by either the client or target. If the client does not support **rpc_c_authn_dce_secret**, then choose **rpc_c_authn_none** and **rpc_c_protect_level_none**. Use the protection level and authentication service thus derived to secure the association between this client and target.

Securing the Binding Handle to the Target

The DCE-CIOP protocol engine acquires an **rpc_binding_handle** to the target using its normal procedure. The **DCE_CIOP** sets authentication and authorization information on that binding handle with the **rpc_binding_set_auth_info()** call using data from the **IOR** profile security components in the following way:

- The target security name string from the **TAG_SEC_NAME** component (or NUL, if the component is not present) is passed to **rpc_binding_set_auth_info()** via the **server_princ_name** parameter.
- If the **TAG_ASSOCIATION_OPTIONS** component is present in the **IOR**, see “Deriving DCE Security Parameters from Association Options” on page 3-112 above to select a DCE RPC protection level and authentication service for this invocation.

  If the **TAG_ASSOCIATION_OPTIONS** component is not present in the IOR, see “Behavior When TAG_ASSOCIATION_OPTIONS Not Present” on page 3-112 above to select a DCE RPC protection level and authentication service for this invocation.

The selected protection level is passed to **rpc_binding_set_auth_info()** via the **protect_level** parameter. The selected authentication service is passed via the **authn_svc** parameter to **rpc_binding_set_auth_info()**.

- The **auth_identity** parameter is set to NULL to use the DCE default login context.
- The authorization service identifier from the **authorization_service** field of the **DCESecurityMechanismInfo component_data** is mapped to the corresponding DCE RPC authorization service identifier (using Table 3-15 on page 3-110) which is then passed via the **authz_svc** parameter.

After a successful call to **rpc_binding_set_auth_info()**, the authenticated binding handle will be used by the DCE-CIOP protocol engine to make secure requests.

References A

A.1 List of References

Note that these references are to definitions which are sometimes a set of document.

[1] CORBA/IIOP 2.2.

[2] Common Secure IIOP Request for Proposals (orb/96-01-03)

[3] CORBA Time Service, Chapter 16 of CORBAservices specification, also available at the URL http://www.omg.org/docs/formal/97-02-22.pdf

[4] IETF RFC 1779 A String Representation of Distinguished Names. March 1995.

[5] *X/Open Application Environment Specification for Distributed Computing*.

[6] X/Open Preliminary Specification *X/Open DCE: Authentication and Security Services.*

[7] X/OPEN *CAE Specification C309*

[8] OSF *AES/Distributed Computing RPC Volume*.

[9] *OSF DCE 1.1 Application Development Reference*

[10] The ECMA GSS-API mechanism specified in ECMA-235. See also related standard ECMA-219 (Authentication and Privilege Attribute Security Application with related key distribution functions)

[11] GSS-APIThe Generic Security Services API as defined in IETF RFC 1508 (September 1993) and X/Open P308.An update to RFC 1508 has been produced by the IETF cat group.

[12] The IETF GSS Kerberos V5 definition which specifies details of the use of Kerberos V5 with GSS-API. It includes updates to RFC 1510 e.g. how to carry delegation information. It is specified in RFC 1964.

[13] The Kerberos V5 mechanism as defined in IETF RFC 1510 (September 1993).

[14] The ORB Portability Specification - CORBA V2.3 Chapter 9..

[15] Open Distributed Processing - Reference Model Parts 1 through 3, OMG doc #om/96-10-02, 03, 04.

[16] The SESAME gss-api mechanism. This is a subset of the ECMA GSS Mechanism and is specified in draft-ietf-cat-sesamemech-00.txt.

[17] The SESAME V4 Overview. This can be found via the web at www.esat.kuleuven.ac.be/cosic/sesame.html

[18] *John G. Fletcher, "Serial Link Protocol Design: A Critique of the X.25* Standard, Level 2," Proceedings of SIGCOMM '84, ACM SIGCOMM, pp.26-33, June 6-8, 1984.

[19] Simple negotiation GSS-API mechanism as defined in draft-ietf-cat-snego-02.txt.

[20] The Simple Public-Key GSS-API Mechanism (SPKM). Internet Draft draft-ietf-cat-spkmgss-06.txt Jan. 1996.

[21] Secure Socket Layer [ftp://ierf.cnsi.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt]

[22] ISO/IEC 9594-8, "Information Technology - Open Systems Interconnection - The Directory: Authentication Framework", CCITT/ITU Recommendation X.509, 1993.

[23] The extended gss-api supporting access control and delegation extensions defined in draft-ietf-cat-xgssapi-acc-cntrl-00.txt. This interface is also defined in the ECMA GSS-API Mechanism standard - ECMA-235

Consolidated OMG IDL — B

B.1 Introduction

The OMG IDL for CORBA security is split into modules as follows:

- A module containing the common data types used by all security modules.
- A module for application interfaces for each Security Functionality Levels 1 and 2.
- A module for Security Level 2 security policy administration.
- A module for non-repudiation, including the non-repudiation policy administration interface.
- A module for the Replaceable Security Service, as described in Section 2.5, “Implementor’s Security Interfaces,” on page 2-143.
- A module for elements of the SECure Inter Orb Protocol (SECIOP)l.
- A module for elements of the SSL Protocol.
- A module for elements related to Security that are added to the DCE_CIOP Security module.

B.2 General Security Data Module

This subsection defines the OMG IDL for security data types common to the other security modules, which is the module **Security**. The **Security** module depends on the **TimeBase** module and the **CORBA** module.

```
#if !defined(_SECURITY_IDL_)
#define _SECURITY_IDL_
#include <orb.idl>
#include <TimeBase.idl>
#pragma prefix "omg.org"

module Security {
```

```
#     pragma version Security 1.5

      typedef string                       SecurityName;
      typedef sequence <octet>             Opaque;

      // Constant declarations for Security Service Options

      const CORBA::ServiceOption SecurityLevel1 = 1;
      const CORBA::ServiceOption SecurityLevel2 = 2;
      const CORBA::ServiceOption NonRepudiation = 3;
      const CORBA::ServiceOption SecurityORBServiceReady = 4;
      const CORBA::ServiceOption SecurityServiceReady = 5;
      const CORBA::ServiceOption ReplaceORBServices = 6;
      const CORBA::ServiceOption ReplaceSecurityServices = 7;
      const CORBA::ServiceOption StandardSecureInteroperability = 8;
      const CORBA::ServiceOption DCESecureInteroperability = 9;

      // Service options for Common Secure Interoperability

      const CORBA::ServiceOption CommonInteroperabilityLevel0 = 10;
      const CORBA::ServiceOption CommonInteroperabilityLevel1 = 11;
      const CORBA::ServiceOption CommonInteroperabilityLevel2 = 12;

      // Security mech types supported for secure association
      const CORBA::ServiceDetailType SecurityMechanismType = 1;

      // privilege types supported in standard access policy
      const CORBA::ServiceDetailType SecurityAttribute = 2;

      // extensible families for standard data types

      struct ExtensibleFamily {
            unsigned short          family_definer;
            unsigned short          family;
      };

      // security attributes

      typedef unsigned long         SecurityAttributeType;

      // other attributes; family = 0

      const SecurityAttributeType           AuditId = 1;
      const SecurityAttributeType           AccountingId = 2;
      const SecurityAttributeType           NonRepudiationId = 3;

      // privilege attributes; family = 1

      const SecurityAttributeType           _Public = 1;
      const SecurityAttributeType           AccessId = 2;
      const SecurityAttributeType           PrimaryGroupId = 3;
      const SecurityAttributeType           GroupId = 4;
      const SecurityAttributeType           Role = 5;
      const SecurityAttributeType           AttributeSet = 6;
      const SecurityAttributeType           Clearance = 7;
const SecurityAttributeType              Capability = 8;

struct AttributeType {
      ExtensibleFamily                   attribute_family;
      SecurityAttributeType              attribute_type;
};

typedef sequence<AttributeType>              AttributeTypeList;

struct SecAttribute {
      AttributeType                attribute_type;
      Opaque                       defining_authority;
      Opaque                       value;
      // the value of this attribute can be
      // interpreted only with knowledge of type
};

typedef sequence <SecAttribute> AttributeList;

// Authentication return status

enum AuthenticationStatus {
      SecAuthSuccess,
      SecAuthFailure,
      SecAuthContinue,
      SecAuthExpired
};

// Association return status

enum AssociationStatus {
      SecAssocSuccess,
      SecAssocFailure,
      SecAssocContinue
};

// Authentication method
typedef unsigned long AuthenticationMethod;

typedef sequence<AuthenticationMethod> AuthenticationMethodList;

// Credential types which can be set as Current default

enum CredentialType {
      SecInvocationCredentials,
      SecNRCredentials
};

enum InvocationCredentialsType {
      SecOwnCredentials,
      SecReceivedCredentials
};

// Declarations related to Rights
```

```
struct Right {
      ExtensibleFamily            rights_family;
      string                      right;
};

typedef sequence <Right> RightsList;

enum RightsCombinator {
      SecAllRights,
      SecAnyRight
};

// Delegation related

enum DelegationState {
      SecInitiator,
      SecDelegate
};

enum DelegationDirective {
      Delegate,
      NoDelegate
};

// pick up from TimeBase

typedef TimeBase::UtcT            UtcT;
typedef TimeBase::IntervalT       IntervalT;
typedef TimeBase::TimeT           TimeT;

// Security features available on credentials.

enum SecurityFeature {
      SecNoDelegation,
      SecSimpleDelegation,
      SecCompositeDelegation,
      SecNoProtection,
      SecIntegrity,
      SecConfidentiality,
      SecIntegrityAndConfidentiality,
      SecDetectReplay,
      SecDetectMisordering,
      SecEstablishTrustInTarget,
      SecEstablishTrustInClient
};

// Quality of protection which can be specified
// for an object reference and used to protect messages

enum QOP {
      SecQOPNoProtection,
      SecQOPIntegrity,
      SecQOPConfidentiality,
      SecQOPIntegrityAndConfidentiality
};
```

```
// Type of SecurityContext

enum SecurityContextType {
      SecClientSecurityContext,
      SecServerSecurityContext
};

// Operational State of a Security Context

enum SecurityContextState {
      SecContextInitialized,
      SecContextContinued,
      SecContextClientEstablished,
      SecContextEstablished,
      SecContextEstablishExpired,
      SecContextExpired,
      SecContextInvalid
};

// For use with SecurityReplaceable

struct OpaqueBuffer {
      Opaque                    buffer;
      unsigned long             startpos;
      unsigned long             endpos;
      // startpos <= endpos
      // OpaqueBuffer is said to be empty if startpos == endpos
};

// Association options which can be administered
// on secure invocation policy and used to
// initialize security context

typedef unsigned short          AssociationOptions;

const AssociationOptions NoProtection = 1;
const AssociationOptions Integrity = 2;
const AssociationOptions Confidentiality = 4;
const AssociationOptions DetectReplay = 8;
const AssociationOptions DetectMisordering = 16;
const AssociationOptions EstablishTrustInTarget = 32;
const AssociationOptions EstablishTrustInClient = 64;
const AssociationOptions NoDelegation = 128;
const AssociationOptions SimpleDelegation = 256;
const AssociationOptions CompositeDelegation = 512;

// Flag to indicate whether association options being
// administered are the “required” or “supported” set

enum RequiresSupports {
      SecRequires,
      SecSupports
};

// Direction of communication for which
// secure invocation policy applies

enum CommunicationDirection {
    SecDirectionBoth,
    SecDirectionRequest,
    SecDirectionReply
};

// security association mechanism type

typedef string                 MechanismType;

typedef sequence<MechanismType> MechanismTypeList;

struct SecurityMechanismData {
    MechanismType              mechanism;
    Opaque                     security_name;
    AssociationOptions         options_supported;
    AssociationOptions         options_required;
};

typedef sequence<SecurityMechanismData>SecurityMechanismDataList;

// AssociationOptions-Direction pair

struct OptionsDirectionPair {
    AssociationOptions         options;
    CommunicationDirection direction;
};

typedef sequence <OptionsDirectionPair> OptionsDirectionPairList;

// Delegation mode which can be administered

enum DelegationMode {
    SecDelModeNoDelegation,                // i.e. use own credentials
    SecDelModeSimpleDelegation,            // delegate received credentials
    SecDelModeCompositeDelegation          // delegate both;
};

// Association options supported by a given mech type

struct MechandOptions {
    MechanismType              mechanism_type;
    AssociationOptions         options_supported;
};

typedef sequence <MechandOptions>           MechandOptionsList;

// Attribute of the SecurityLevel2::EstablishTrustPolicy

struct EstablishTrust {
    boolean trust_in_client;
    boolean trust_in_target;
};
```

```
// Audit

typedef unsigned long                       AuditChannelId;

typedef unsigned short                      EventType;

const EventType         AuditAll = 0;
const EventType         AuditPrincipalAuth = 1;
const EventType         AuditSessionAuth = 2;
const EventType         AuditAuthorization = 3;
const EventType         AuditInvocation = 4;
const EventType         AuditSecEnvChange = 5;
const EventType         AuditPolicyChange = 6;
const EventType         AuditObjectCreation = 7;
const EventType         AuditObjectDestruction = 8;
const EventType         AuditNonRepudiation = 9;

enum DayOfTheWeek {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

enum AuditCombinator {
    SecAllSelectors,
    SecAnySelector
};

struct AuditEventType {
    ExtensibleFamily        event_family;
    EventType               event_type;
};
typedef sequence <AuditEventType>           AuditEventTypeList;

typedef unsigned long                       SelectorType;

const SelectorType      InterfaceName = 1;
const SelectorType      ObjectRef = 2;
const SelectorType      Operation = 3;
const SelectorType      Initiator = 4;
const SelectorType      SuccessFailure = 5;
const SelectorType      Time = 6;
const SelectorType      DayOfWeek = 7;

// values defined for audit_needed and audit_write are:
// InterfaceName: CORBA::RepositoryId
// ObjectRef: object reference
// Operation: op_name
// Initiator: Credentials
// SuccessFailure: boolean
// Time: utc time on audit_write; time picked up from
//              environment in audit_needed if required
// DayOfWeek: DayOfTheWeek

struct SelectorValue {
    SelectorType        selector;
    any                 value;
};
```

```
    typedef sequence <SelectorValue>          SelectorValueList;

    // Constant declaration for valid Security Policy Types

    // General administrative policies
    const CORBA::PolicyType SecClientInvocationAccess = 1;
    const CORBA::PolicyType SecTargetInvocationAccess = 2;
    const CORBA::PolicyType SecApplicationAccess = 3;
    const CORBA::PolicyType SecClientInvocationAudit = 4;
    const CORBA::PolicyType SecTargetInvocationAudit = 5;
    const CORBA::PolicyType SecApplicationAudit = 6;
    const CORBA::PolicyType SecDelegation = 7;
    const CORBA::PolicyType SecClientSecureInvocation = 8;
    const CORBA::PolicyType SecTargetSecureInvocation = 9;
    const CORBA::PolicyType SecNonRepudiation = 10;

    // Policies used to control attributes of a binding to a target
    const CORBA::PolicyType SecMechanismsPolicy = 12;
    const CORBA::PolicyType SecInvocationCredentialsPolicy = 13;
    const CORBA::PolicyType SecFeaturePolicy = 14; // obsolete
    const CORBA::PolicyType SecQOPPolicy = 15;

    const CORBA::PolicyType SecDelegationDirectivePolicy = 38;
    const CORBA::PolicyType SecEstablishTrustPolicy = 39;
};
#endif /* _SECURITY_IDL_ */
```

B.3 Application Interfaces - Level 1

This subsection defines those interfaces available to application objects using only Security Functionality Level 1, and consists of a single module, **SecurityLevel1**. This module depends on the **CORBA** module, and on the **Security** module.

```
#if !defined(_SECURITY_LEVEL_1_IDL_)
#define _SECURITY_LEVEL_1_IDL_
#include <Security.idl>
#pragma prefix "omg.org"

module SecurityLevel1 {

#   pragma version SecurityLevel1 1.5
    interface Current : CORBA::Current {// Locality Constrained

        // thread specific operations

        Security::AttributeList get_attributes (
            in   Security::AttributeTypeList        attributes
        );
    };
};
#endif /* _SECURITY_LEVEL_1_IDL_ */
```

B.4 Application Interfaces - Level 2

This subsection defines the interfaces available to applications using Security Functionality Level 2, all of which are declared in the **SecurityLevel2** module. This module depends on the **CORBA, SecurityLevel1** and **Security** modules. The interfaces are described in Section 2.3, “Application Developer’s Interfaces,” on page 2-71.

```
#if !defined(_SECURITY_LEVEL_2_IDL_)
#define _SECURITY_LEVEL_2_IDL_
#include <SecurityLevel1.idl>
#pragma prefix "omg.org"

module SecurityLevel2 {

#    pragma version SecurityLevel2 1.5

     // Forward declaration of interfaces
     interface PrincipalAuthenticator;
     interface Credentials;
     interface Current;

// Interface PrincipalAuthenticator
     interface PrincipalAuthenticator {             // Locality Constrained
#         pragma version PrincipalAuthenticator 1.5

          Security::AuthenticationMethodList
          get_supported_authen_methods(
               in       Security::MechanismType              mechanism
          );

          Security::AuthenticationStatus authenticate (
               in       Security::AuthenticationMethod       method,
               in       Security::MechanismType              mechanism,
               in       Security::SecurityName               security_name,
               in       Security::Opaque                     auth_data,
               in       Security::AttributeList              privileges,
               out      Credentials                          creds,
               out      Security::Opaque                     continuation_data,
               out      Security::Opaque                     auth_specific_data
          );

          Security::AuthenticationStatus continue_authentication (
               in       Security::Opaque                     response_data,
               in       Credentials                          creds,
               out      Security::Opaque                     continuation_data,
               out      Security::Opaque                     auth_specific_data
          );
     };

     // Interface Credentials
     interface Credentials {              // Locality Constrained
#         pragma version Credentials 1.5
```

```
        Credentials copy ();

        void destroy();

        readonly attribute Security::InvocationCredentialsTypecredentials_type;

        readonly attribute Security::AuthenticationStatus  authentication_state;

        readonly attribute Security::MechanismType mechanism;

        attribute Security::AssociationOptions      accepting_options_supported;

        attribute Security::AssociationOptions      accepting_options_required;

        attribute Security::AssociationOptions      invocation_options_supported;

        attribute Security::AssociationOptions      invocation_options_required;

        boolean get_security_feature(
            in      Security::CommunicationDirection        direction,
            in      Security::SecurityFeature               feature
        );

        boolean set_privileges (
            in      boolean                         force_commit,
            in      Security::AttributeList         requested_privileges,
            out     Security::AttributeList         actual_privileges
        );

        Security::AttributeList get_attributes (
            in      Security::AttributeTypeList     attributes
        );

        boolean is_valid (
            out     Security::UtcT                  expiry_time
        );

        boolean refresh(
            in      Security::Opaque                refresh_data
        );
    };

    typedef sequence <Credentials>              CredentialsList;

    interface ReceivedCredentials : Credentials { // Locality Constrained

#       pragma version ReceivedCredentials 1.5

        readonly attribute Credentials          accepting_credentials;

        readonly attribute Security::AssociationOptionsassociation_options_used;

        readonly attribute Security::DelegationState    delegation_state;

        readonly attribute Security::DelegationMode     delegation_mode;
```

```
};

// RequiredRights Interface

interface RequiredRights{
    void get_required_rights(
        in      Object                          obj,
        in      CORBA::Identifier               operation_name,
        in      CORBA::RepositoryId             interface_name,
        out     Security::RightsList            rights,
        out     Security::RightsCombinator      rights_combinator
    );

    void set_required_rights(
        in      CORBA::Identifier               operation_name,
        in      CORBA::RepositoryId             interface_name,
        in      Security::RightsList            rights,
        in      Security::RightsCombinator      rights_combinator
    );
};

// interface audit channel
interface AuditChannel {                    // Locality Constrained

    void audit_write (
        in      Security::AuditEventType        event_type,
        in      CredentialsList                 creds,
        in      Security::UtcT                  time,
        in      Security::SelectorValueList     descriptors,
        in      Security::Opaque                event_specific_data
    );

    readonly attribute Security::AuditChannelId        audit_channel_id;
};

// interface for Audit Decision

interface AuditDecision {                       // Locality Constrained

    boolean audit_needed (
        in      Security::AuditEventType        event_type,
        in      Security::SelectorValueList     value_list
    );

    readonly attribute AuditChannel audit_channel;
};

interface AccessDecision {                      // Locality Constrained

    boolean access_allowed (

        in      SecurityLevel2::CredentialsList     cred_list,
        in      Object                              target,
        in      CORBA::Identifier                   operation_name,
        in      CORBA::Identifier                   target_interface_name
        );
    };

    // Policy interfaces to control bindings

    interface QOPPolicy : CORBA::Policy {                // Locality Constrained
        readonly attribute Security::QOP                     qop;
    };

    interface MechanismPolicy : CORBA::Policy {  // Locality Constrained
        readonly attribute Security::MechanismTypeList          mechanisms;
    };

    interface InvocationCredentialsPolicy : CORBA::Policy {
                                                         // Locality Constrained
        readonly attribute CredentialsList                   creds;
    };

    interface EstablishTrustPolicy : CORBA::Policy { // Locality Constrained
        readonly attribute Security::EstablishTrust          trust;
    };

    interface DelegationDirectivePolicy : CORBA::Policy {
                                                         // Locality Constrained
        readonly attribute Security::DelegationDirective
                                                 delegation_directive;
    };

    // Interface Current derived from SecurityLevel1::Current  providing
    // additional operations on Current at this security level.
    // This is implemented by the ORB

    interface Current : SecurityLevel1::Current {        // Locality Constrained

#       pragma version Current 1.5

    // Thread specific

        readonly attribute ReceivedCredentials received_credentials;

        void set_credentials (
            in      Security::CredentialType         cred_type,
            in      CredentialsList                  creds,
            in      Security::DelegationDirective    del
        );

        CredentialsList get_credentials (
            in      Security::CredentialType     cred_type
        );

        CORBA::Policy get_policy (
            in      CORBA::PolicyType            policy_type
        );

        void remove_own_credentials(
            in      Credentials                     creds
        );

    // Process/Capsule/ORB Instance specific operations

        readonly attribute Security::MechandOptionsListsupported_mechanisms;

        readonly attribute CredentialsList          own_credentials;

        readonly attribute RequiredRights           required_rights_object;
        readonly attribute PrincipalAuthenticator
                                                    principal_authenticator;
        readonly attribute AccessDecision           access_decision;
        readonly attribute AuditDecision            audit_decision;

        // Security mechanism data for a given target
        Security::SecurityMechanismDataList get_security_mechanisms (
            in      Object                          obj_ref
        );
    };
};

#endif /* _SECURITY_LEVEL_2_IDL_ */
```

B.5 Security Administration Interfaces

This section covers interfaces concerned with querying and modifying security policies, and comprises the module **SecurityAdmin**. The **SecurityAdmin** module depends on **CORBA**, **Security**, and **SecurityLevel2** modules. The interfaces are described in Section 2.4, “Administrator’s Interfaces,” on page 2-116. There are related interfaces for finding domain managers and policies. They are to be found in the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*.

```
#if !defined(_SECURITY_ADMIN_IDL_)
#define _SECURITY_ADMIN_IDL_
#include <SecurityLevel2.idl>
#pragma prefix "omg.org"

module SecurityAdmin {

#   pragma version SecurityAdmin 1.5

    // interface AccessPolicy
    interface AccessPolicy : CORBA::Policy {
#       pragma version AccessPolicy 1.5

        Security::RightsList get_effective_rights (
            in      Security::AttributeList             attrib_list,
            in      Security::ExtensibleFamily          rights_family
        );

        Security::RightsList get_all_effective_rights(
                in        Security::AttributeList               attrib_list
            );
        };

        // interface DomainAccessPolicy
        interface DomainAccessPolicy : AccessPolicy {
#           pragma version DomainAccessPolicy 1.5

            void grant_rights(
                in        Security::SecAttribute                priv_attr,
                in        Security::DelegationState             del_state,
                in        Security::RightsList                  rights
            );

            void revoke_rights(
                in        Security::SecAttribute                priv_attr,
                in        Security::DelegationState             del_state,
                in        Security::RightsList                  rights
            );

            void replace_rights (
                in        Security::SecAttribute                priv_attr,
                in        Security::DelegationState             del_state,
                in        Security::RightsList                  rights
            );

            Security::RightsList get_rights (
                in        Security::SecAttribute                priv_attr,
                in        Security::DelegationState             del_state,
                in        Security::ExtensibleFamily            rights_family
            );

            Security::RightsList get_all_rights(
                in        Security::SecAttribute                priv_attr,
                in        Security::DelegationState             del_state
            );
        };

        // interface AuditPolicy
        interface AuditPolicy : CORBA::Policy {
#           pragma version AuditPolicy 1.5

            void set_audit_selectors (
                in        CORBA::RepositoryId                   object_type,
                in        Security::AuditEventTypeList          events,
                in        Security::SelectorValueList           selectors,
                in        Security::AuditCombinator             audit_combinator
            );

            void clear_audit_selectors (
                in        CORBA::RepositoryId                   object_type,
                in        Security::AuditEventTypeList          events
            );
        void replace_audit_selectors (
            in      CORBA::RepositoryIdf                object_type,
            in      Security::AuditEventTypeList        events,
            in      Security::SelectorValueList         selectors,
            in      Security::AuditCombinator           audit_combinator
        );

        void get_audit_selectors (
            in      CORBA::RepositoryId                 object_type,
            in      Security::AuditEventType            event_type
            out     Security::SelectorValueList         selectors,
            out     Security::AuditCombinator           audit_combinator
        );

        void set_audit_channel (
            in      Security::AuditChannelId            audit_channel_id
        );
    };

    // interface SecureInvocationPolicy
    interface SecureInvocationPolicy : CORBA::Policy {
#       pragma version SecureInvocationPolicy 1.5

        void set_association_options(
            in      CORBA::RepositoryId                 object_type,
            in      Security::RequiresSupports          requires_supports,
            in      Security::CommunicationDirection    direction,
            in      Security::AssociationOptions        options
        );

        Security::AssociationOptions get_association_options(
            in      CORBA::RepositoryId                 object_type,
            in      Security::RequiresSupports          requires_supports,
            in      Security::CommunicationDirection    direction
        );
    };

    // interface DelegationPolicy
    interface DelegationPolicy : CORBA::Policy {
#       pragma version DelegationPolicy 1.5

        void set_delegation_mode(
            in      CORBA::RepositoryId                 object_type,
            in      Security::DelegationMode            mode
        );

        Security::DelegationMode get_delegation_mode(
            in      CORBA::RepositoryId                 object_type
        );
    };
};

#endif /* _SECURITY_ADMIN_IDL_ */
```

B.6 Interfaces for Non-repudiation

This subsection defines the optional application interface for non-repudiation. This module depends on **SecurityLevel2** and **CORBA** modules. The interfaces are described in Section 2.1.7, "Non-repudiation," on page 2-18.

```
#if !defined(_NR_SERVICE_IDL_)
#define _NR_SERVICE_IDL_
#include <SecurityLevel2.idl>
#pragma prefix "omg.org"

module NRService  {

#     pragma version NRService 1.5

      typedef Security::MechanismType              NRMech;
      typedef Security::ExtensibleFamily           NRPolicyId;

      enum EvidenceType {
            SecProofofCreation,
            SecProofofReceipt,
            SecProofofApproval,
            SecProofofRetrieval,
            SecProofofOrigin,
            SecProofofDelivery,
            SecNoEvidence     // used when request-only token desired
      };

      enum NRVerificationResult {
            SecNRInvalid,
            SecNRValid,
            SecNRConditionallyValid
      };

      // the following are used for evidence validity duration
      typedef unsigned long          DurationInMinutes;

      const DurationInMinutes DurationHour   = 60;
      const DurationInMinutes DurationDay    = 1440;
      const DurationInMinutes DurationWeek   = 10080;
      const DurationInMinutes DurationMonth = 43200;// 30 days;
      const DurationInMinutes DurationYear   = 525600;//365 days;

      typedef long TimeOffsetInMinutes;

      struct NRPolicyFeatures {
            NRPolicyId              policy_id;
            unsigned long           policy_version;
            NRMech                  mechanism;
      };
      typedef sequence <NRPolicyFeatures> NRPolicyFeaturesList;

      // features used when generating requests
      struct RequestFeatures {
```

```
    NRPolicyFeatures          requested_policy;
    EvidenceType              requested_evidence;
    string                    requested_evidence_generators;
    string                    requested_evidence_recipients;
    boolean                   include_this_token_in_evidence;
};

struct EvidenceDescriptor {
    EvidenceType              evidence_type;
    DurationInMinutes         evidence_validity_duration;
    boolean                   must_use_trusted_time;
};
typedef sequence <EvidenceDescriptor> EvidenceDescriptorList;

struct AuthorityDescriptor {
    string                    authority_name;
    string                    authority_role;
    TimeOffsetInMinutes       last_revocation_check_offset;
            // may be >0 or <0; add this to evid. gen. time to
            // get latest time at which mech. will check to see
            // if this authority's key has been revoked.
};
typedef sequence <AuthorityDescriptor> AuthorityDescriptorList;

struct MechanismDescriptor {
    NRMech                        mech_type;
    AuthorityDescriptorList       authority_list;
    TimeOffsetInMinutes           max_time_skew;
            // max permissible difference between evid. gen. time
            // and time of time service countersignature
            // ignored if trusted time not reqd.
};
typedef sequence <MechanismDescriptor> MechanismDescriptorList;

interface NRCredentials : SecurityLevel2::Credentials{

    boolean set_NR_features (
        in      NRPolicyFeaturesList        requested_features,
        out     NRPolicyFeaturesList        actual_features
    );

    NRPolicyFeaturesList get_NR_features ();

    void generate_token (
        in      Security::Opaque            input_buffer,
        in      EvidenceType                generate_evidence_type,
        in      boolean                     include_data_in_token,
        in      boolean                     generate_request,
        in      RequestFeatures             request_features,
        in      boolean                     input_buffer_complete,
        out     Security::Opaque            nr_token,
        out     Security::Opaque            evidence_check
    );
```

```
        NRVerificationResult verify_evidence (
            in      Security::Opaque            input_token_buffer,
            in      Security::Opaque            evidence_check,
            in      boolean                     form_complete_evidence,
            in      boolean                     token_buffer_complete,
            out     Security::Opaque            output_token,
            out     Security::Opaque            data_included_in_token,
            out     boolean                     evidence_is_complete,
            out     boolean                     trusted_time_used,
            out     Security::TimeT             complete_evidence_before,
            out     Security::TimeT             complete_evidence_after
    );

        void get_token_details (
            in      Security::Opaque            token_buffer,
            in      boolean                     token_buffer_complete,
            out     string                      token_generator_name,
            out     NRPolicyFeatures            policy_features,
            out     EvidenceType                evidence_type,
            out     Security::UtcT              evidence_generation_time,
            out     Security::UtcT              evidence_valid_start_time,
            out     DurationInMinutes           evidence_validity_duration,
            out     boolean                     data_included_in_token,
            out     boolean                     request_included_in_token,
            out     RequestFeatures             request_features
        );

        boolean form_complete_evidence (
            in      Security::Opaque            input_token,
            out     Security::Opaque            output_token,
            out     boolean                     trusted_time_used,
            out     Security::TimeT             complete_evidence_before,
            out     Security::TimeT             complete_evidence_after
        );
    };

    interface NRPolicy : CORBA::Policy{

        void get_NR_policy_info   (
            out     Security::ExtensibleFamily
                                                NR_policy_id,
            out     unsigned long               policy_version,
            out     Security::TimeT             policy_effective_time,
            out     Security::TimeT             policy_expiry_time,
            out     EvidenceDescriptorList      supported_evidence_types,
            out     MechanismDescriptorList     supported_mechanisms
        );

        boolean set_NR_policy_info (
            in      MechanismDescriptorList     requested_mechanisms,
            out     MechanismDescriptorList     actual_mechanisms
        );
    };
};
#endif /* _NR_SERVICE_IDL_ */
```

B.7 Security Replaceable Service Interfaces

This section defines the IDL interfaces to the Security objects, which should be replaced if there is a requirement to replace the Security services used for security associations (i.e., the **Vault** and **Security Context**). The IDL provided here is for those interfaces that have not already been covered by the **SecurityLevel2** module. This section comprises the module **SecurityReplaceable**. This module depends on the **CORBA**, **Security**, and **SecurityLevel2** modules. The interfaces are described in Section 2.5, “Implementor’s Security Interfaces,” on page 2-143.

```
#if !defined(_SECURITY_REPLACEABLE_IDL_)
#define _SECURITY_REPLACEABLE_IDL_
#include <SecurityLevel2.idl>
#pragma prefix "omg.org"


module SecurityReplaceable {

#   pragma version SecurityReplacable 1.5

    interface SecurityContext;
    interface ClientSecurityContext;
    interface ServerSecurityContext;

    interface Vault {                       // Locality Constrained

#       pragma version Vault 1.5

        Security::AuthenticationMethodList
        get_supported_authen_methods(
            in      Security::MechanismType             mechanism;
        );

        Security::AuthenticationStatus acquire_credentials(
            in      Security::AuthenticationMethod      method,
            in      Security::MechanismType             mechanism,
            in      Security::SecurityName              security_name,
            in      Security::Opaque                    auth_data,
            in      Security::AttributeList             privileges,
            out     SecurityLevel2::Credentials         creds,
            out     Security::Opaque                    continuation_data,
            out     Security::Opaque                    auth_specific_data
        );

        Security::AuthenticationStatus continue_credentials_acquisition(
            in      Security::Opaque                    response_data,
            in      SecurityLevel2::Credentials         creds,
            out     Security::Opaque                    continuation_data,
            out     Security::Opaque                    auth_specific_data
        );

        Security::AssociationStatus init_security_context (
            in      SecurityLevel2::Credentials         creds,
            in      Security::SecurityName              target_security_name,
          in        Object                             target,
          in        Security::DelegationMode           delegation_mode,
          in        Security::OptionsDirectionPairList association_options,
          in        Security::MechanismType            mechanism,
          in        Security::Opaque                   mech_data, //from IOR
          in        Security::Opaque                   chan_binding,
          out       Security::OpaqueBuffer             security_token,
          out       ClientSecurityContext              security_context
      );

      Security::AssociationStatus accept_security_context (
          in        SecurityLevel2::CredentialsList    creds_list,
          in        Security::Opaque                   chan_bindings,
          in        Security::OpaqueBuffer             in_token,
          out       Security::OpaqueBuffer             out_token,
          out       ServerSecurityContext              security_context
      );

      Security::MechandOptionsList get_supported_mechs ();
  };

  interface SecurityContext {             // Locality Constrained

#     pragma version SecurityContext 1.5

      readonly attribute Security::SecurityContextType  context_type;
      readonly attribute Security::SecurityContextState context_state;
      readonly attribute Security::MechanismType        mechanism;

      readonly attribute boolean                 supports_refresh;

      readonly attribute Security::Opaque        chan_binding;

      readonly attribute SecurityLevel2::ReceivedCredentials
      received_credentials;

      Security::AssociationStatus continue_security_context (
          in        Security::OpaqueBuffer          in_token,
          out       Security::OpaqueBuffer          out_token
      );

      void protect_message (
          in        Security::OpaqueBuffer          message,
          in        Security::QOP                   qop,
          out       Security::OpaqueBuffer          text_buffer,
          out       Security::OpaqueBuffer          token
      );

      boolean reclaim_message (
          in        Security::OpaqueBuffer          text_buffer,
          in        Security::OpaqueBuffer          token,
          out       Security::QOP                   qop,
          out       Security::OpaqueBuffer          message
      );
```

```
        boolean is_valid (
            out     Security::UtcT                  expiry_time
        );

        boolean refresh_security_context (
            in      Security::Opaque                refresh_data,
            out     Security::OpaqueBuffer          out_token
        );

        boolean process_refresh_token (
            in      Security::OpaqueBuffer          refresh_token
        );

        boolean discard_security_context (
            in      Security::Opaque                discard_data,
            out     Security::OpaqueBuffer          out_token
        );

        boolean process_discard_token (
            in      Security::OpaqueBuffer          discard_token,
        );
    };

    interface ClientSecurityContext : SecurityContext { // Locality Constrained
        readonly attribute Security::AssociationOptions association_options_used;
        readonly attribute Security::DelegationMode       delegation_mode;
        readonly attribute Security::Opaque               mech_data;
        readonly attribute SecurityLevel2::Credentials    client_credentials;
        readonly attribute Security::AssociationOptionsserver_options_supported;
        readonly attribute Security::AssociationOptionsserver_options_required;
        readonly attribute Security::Opaque            server_security_name;
    };

    interface ServerSecurityContext : SecurityContext { // Locality Constrained
        readonly attribute Security::AssociationOptions association_options_used;
        readonly attribute Security::DelegationMode       delegation_mode;
        readonly attribute SecurityLevel2::Credentials    server_credentials;
        readonly attribute Security::AssociationOptions server_options_supported;
        readonly attribute Security::AssociationOptionsserver_options_required;
        readonly attribute Security::Opaque            server_security_name;
    };
};

#endif /* _SECURITY_REPLACEABLE_IDL_ */
```

B.8 Secure Inter-ORB Protocol (SECIOP)

The **SECIOP** module holds structure declarations related to the layout of message fields in the Secure Inter-ORB protocol. This module depends on the **IOP** and **Security** modules.

```
#if !defined(_SECIOP_IDL_)
#define _SECIOP_IDL
```

```
#include <IOP.idl>
#include <Security.idl>
#pragma prefix "omg.org"


module SECIOP {

    const IOP::ComponentId TAG_GENERIC_SEC_MECH = 22;

    const IOP::ComponentId TAG_ASSOCIATION_OPTIONS = 13;

    const IOP::ComponentId TAG_SEC_NAME = 14;

    struct TargetAssociationOptions{
            Security::AssociationOptions        target_supports;
            Security::AssociationOptions        target_requires;
    };

    struct GenericMechanismInfo {
            sequence <octet>                    security_mechanism_type;
            sequence <octet>                    mech_specific_data;
            sequence <IOP::TaggedComponent>  components;
    };

    enum MsgType {
            MTEstablishContext,
            MTCompleteEstablishContext,
            MTContinueEstablishContext,
            MTDiscardContext,
            MTMessageError,
            MTMessageInContext
    };

    typedef unsigned long long ContextId;

    enum ContextIdDefn {
            CIDClient,
            CIDPeer,
            CIDSender
    };

    struct EstablishContext {
            ContextId               client_context_id;
            sequence <octet>        initial_context_token;
    };

    struct CompleteEstablishContext {
            ContextId               client_context_id;
            boolean                 target_context_id_valid;
            ContextId               target_context_id;
            sequence <octet>        final_context_token;
    };

    struct ContinueEstablishContext {
            ContextId               client_context_id;
            sequence <octet>        continuation_context_token;
};

struct DiscardContext {
            ContextIdDefn           message_context_id_defn;
            ContextId               message_context_id;
            sequence <octet>        discard_context_token;
};

struct MessageError {
            ContextIdDefn           message_context_id_defn;
            ContextId               message_context_id;
            long                    major_status;
            long                    minor_status;
};

enum ContextTokenType {
            SecTokenTypeWrap,
            SecTokenTypeMIC
};

struct MessageInContext {
            ContextIdDefn           message_context_id_defn;
            ContextId               message_context_id;
            ContextTokenType        message_context_type;
            sequence <octet>        message_protection_token;
};

// message_protection_token is obtained by CDR encoding
// the following SequencingHeader followed by the octets of the
// frame data. SequencingHeader + Frame Data is called a
// SequencedDataFrame

struct SequencingHeader {
      octet                   control_state;
      unsigned long           direct_sequence_number;
      unsigned long           reverse_sequence_number;
      unsigned long           reverse_window;
};

typedef sequence <octet> SecurityName;
typedef unsigned short CryptographicProfile;
typedef sequence <CryptographicProfile> CryptographicProfileList;

// Cryptographic profiles for SPKM

const CryptographicProfile          MD5_RSA = 20;
const CryptographicProfile          MD5_DES_CBC = 21;
const CryptographicProfile          DES_CBC = 22;
const CryptographicProfile          MD5_DES_CBC_SOURCE  = 23;
const CryptographicProfile          DES_CBC_SOURCE  = 24;

// Security Mechanism SPKM_1

const IOP::ComponentId              TAG_SPKM_1_SEC_MECH = 15;
```

```
struct SPKM_1 {
      Security::AssociationOptions          target_supports;
      Security::AssociationOptions          target_requires;
      CryptographicProfileList              crypto_profile;
      SecurityName                          security_name;
};

// Security Mechanism SPKM_1

const IOP::ComponentId TAG_SPKM_2_SEC_MECH = 16;

struct SPKM_2 {
      Security::AssociationOptions          target_supports;
      Security::AssociationOptions          target_requires;
      CryptographicProfileList              crypto_profile;
      SecurityName                          security_name;
};

// Cryptographic profiles for GSS Kerberos Protocol

const CryptographicProfile            DES_CBC_DES_MAC = 10;
const CryptographicProfile            DES_CBC_MD5 = 11;
const CryptographicProfile            DES_MAC = 12;
const CryptographicProfile            MD5 = 13;

// Security Mechanism KerberosV5

const IOP::ComponentId TAG_KerberosV5_SEC_MECH = 17;

struct KerberosV5 {
      Security::AssociationOptions          target_supports;
      Security::AssociationOptions          target_requires;
      CryptographicProfileList              crypto_profile;
      SecurityName                          security_name;
};

// Cryptographic profiles for CSI-ECMA Protocol

const CryptographicProfile            FullSecurity = 1;
const CryptographicProfile            NoDataConfidentiality = 2;
const CryptographicProfile            LowGradeConfidentiality = 3;
const CryptographicProfile            AgreedDefault = 5;

// Security Mechanism CSI_ECMA_Secret

const IOP::ComponentId TAG_CSI_ECMA_Secret_SEC_MECH = 18;

struct CSI_ECMA_Secret {
      Security::AssociationOptions          target_supports;
      Security::AssociationOptions          target_requires;
      CryptographicProfileList              crypto_profile;
      SecurityName                          security_name;
};

// Security Mechanism CSI_ECMA_Hybrid
```

```
    const IOP::ComponentId TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;

    struct CSI_ECMA_Hybrid {
        Security::AssociationOptions        target_supports;
        Security::AssociationOptions        target_requires;
        CryptographicProfileList            crypto_profile;
        SecurityName                        security_name;
    };

    // Security Mechanism CSI_ECMA_Public

    const IOP::ComponentId TAG_CSI_ECMA_Public_SEC_MECH = 21;

    struct CSI_ECMA_Public {
        Security::AssociationOptions        target_supports;
        Security::AssociationOptions        target_requires;
        CryptographicProfileList            crypto_profile;
        SecurityName                        security_name;
    };
};
#endif /* _SECIOP_IDL */
```

B.9 SSL

The **SSLIOP** module holds the structure and TAG definitions needed for using SSL as the secure transport under CORBA Security. This module depends on the **Security** and the **IOP** modules.

```
#if !defined(_SSLIOP_IDL)
#define _SSLIOP_IDL
#pragma prefix "omg.org"
#include <IOP.idl>
#include<Security.idl>

module SSLIOP {
    // Security mechanism SSL

    const IOP::ComponentId TAG_SSL_SEC_TRANS = 20;

    struct SSL {
        Security::AssociationOptions        target_supports;
        Security::AssociationOptions        target_requires;
        unsigned short                      port;
    };
};
#endif /* _SSLIOP_IDL */
```

B.10 Secure DCE CIOP

The **DCE_CIOP_Security** module extension holds structures and TAG definitions needed for using DCE-CIOP Security. This module depends on **Security** and **IOP** modules.

```
#if !defined(_DCE_CIOP_SECURITY_IDL)
#define _DCE_CIOP_SECURITY_IDL
#pragma prefix "omg.org"
#include <IOP.idl>
#include <Security.idl>

module DCE_CIOPSecurity {

    const IOP::ComponentId      TAG_DCE_SEC_MECH = 103;

    typedef unsigned short      DCEAuthorization;

    const DCEAuthorization      DCEAuthorizationNone = 0;
    const DCEAuthorization      DCEAuthorizationName = 1;
    const DCEAuthorization      DCEAuthorizationDCE = 2;

    // since consts of type octet are not allowed in IDL the constant
    // values that can be assigned to the authorization_service field
    // in the DCESecurityMechanismInfo is declared as unsigned shorts.
    // when they actually get assigned to the authorization_service field
    // they should be assigned as octets.

    struct DCESecurityMechanismInfo {
        octet                                   authorization_service;
        sequence<IOP::TaggedComponent>          components;
    };
};

#endif /* _DCE_CIOP_SECURITY_IDL */
```

B.11 Values for Standard Data Types

A number of data types in this specification allow an extensible set of values, so the user can add values as required to meet his own security policies. However, if all users defined their own values, portability and interoperability would be seriously restricted.

Therefore, some standard values for certain data types are defined. These include the values that identify:

- Security attributes (privilege and other attribute types)
- Rights families
- Audit event families and types
- Security mechanism types as used in the **IOR** (and **Vault**, etc.)

Rights families and audit event families are defined as an **ExtensibleFamily** type. This has a family definer value registered with OMG and a family id defined by the family definer. Security attribute types also have family definers. Family definers with values **0** - **7** are reserved for OMG. The family value **0** is used for defining standard types (e.g., of security attributes).

B.11.1 Attribute Types

Section 2.3, “Application Developer’s Interfaces,” on page 2-71 defines an attribute structure for privilege and other attributes. This includes:

- A family, as previously described.
- An attribute type. Users may add new attribute types. Two standard OMG families are defined: the family of privilege attributes (**family = 1**), and the family of other attributes (**family = 0**). Types in these families are listed in the following table.
- An optional defining authority. This indicates the authority responsible for defining the value within the attribute type. Some policies demand that multiple sources of values for a given attribute type be supported (e.g., a policy accepting attribute values defined outside the security domain). These policies give rise to a risk of value clashes. The defining authority field is used to separate these values. When not present (i.e., **length = 0**), the value defaults to the name of the authority that issued the attribute.
- An attribute value. The attribute value is defined as a **sequence<octet>**, which someone who understands that attribute type can decipher.

Table B-1 Attribute Values

Attribute	**Type Value**	**Meaning**
Privilege Attributes (family = 1)		**All privilege attributes are used for access control**
Public	1	The principal has no authenticated identity
AccessId	2	The identity of the principal used for access control
PrimaryGroupId	3	The primary group to which the principal belongs
GroupId	4	A group to which the principal belongs
Role	5	A role the principal takes
AttributeSet	6	An identifier for a set of related attributes, which a user or application can obtain
Clearance	7	The principal’s security clearance
Capability	8	A capability
Other Attributes (family = 0)		
AuditId	1	The identity of the principal used for auditing
AccountingId	2	The id of the account to be charged for resource use
NonRepudiationId	3	The id of the principal used for non-repudiation

B.11.2 Rights Families and Values

Administration is simplified by defining rights that provide access to a set of operations, so the administrator only needs to know what rights are required, rather than the semantics of particular operations.

Rights are grouped into families. Only one rights family is defined in this specification. The family definer is OMG (value **0**) and the family id is CORBA (value **1**). Other families may be added by vendors or users.

Three values are specified for the standard CORBA rights family.

Table B-2 CORBA Rights Family Values

Right	Meaning
"**get**"	Used for any operation on the object that does not change its state
"**set**"	For operations on an object that changes its state
"**manage**"	For operations on the attributes of the object, not its state
"**use**"	For operations on an object that may change the overall state of the system, but not the state of the object itself

B.11.3 Audit Event Families and Types

Events, like rights, are grouped into families as defined in Section 2.3, "Application Developer's Interfaces," on page 2-71.

Only one event family is defined in this specification. This has a family definer of OMG (value **0**) and family of SYSTEM (value **1**) and is used for auditing system events. All events of this type are audited by the object security services, or the underlying security services they use. Some of these events must be audited by secure object systems conforming to Security Functionality Level 1 (though in some cases, the event may be audited by underlying security services). Other event types are identified so that, if produced, a standard record is generated, so that audit trails from different systems can more easily be combined. System audit events are specified in the table below.

Table B-3 System Audit Events

Event Name	Value	Whether Mandatory	Meaning and Event Specific Data
AuditPrincipalAuth	1	Yes	Authentication of principals, either via the principal authentication interface or underlying security services
AuditSessionAuth	2	Yes	Security association/peer authentication
AuditAuthorization	3	Yes	Authorization of an object invocation (normally using an Access Decision object)
AuditInvocation	4	No	Object invocation (i.e. the request/reply)
AuditSecEnvChange	5	No	Change to the security environment for this client or object (e.g. override_default_credentials)
AuditPolicyChange	6	Yes	Change to a security policy (using the administrative interfaces in Section 15.6, Administrator's Interfaces)
AuditObjectCreation	7	No	Creation of an object
AuditObjectDestruction	8	No	Destruction of an object
AuditNonRepudiation	9	No	Generation or verification of evidence

Application audit policies are expected to use application audit families.

B.11.4 Security Mechanisms

The security specification allows use of different mechanisms for security associations. These are used in the Interoperable Object Reference and also on the interface to the Vault.

Mechanism ids that are formed by stringifying the integer value of the corresponding mechanism tag value. So, for example the mechanism id of mechanism type SPKM_1 is the string "15", which is the string representation of the mechanism tag value defined in the SECIOP module above as TAG_SPKM_1_SEC_MECH.

Following this rule, the currently defined mechanism ids are listed in the table below.

Table B-4 Mechanism Ids

Mechanism Name	Mechanism Tag	Mech Id	Base Mech
SPKM_1	TAG_SPKM_1_SEC_MECH	**"15"**	SPKM
SPKM_2	TAG_SPKM_2_SEC_MECH	**"16"**	SPKM
KerberosV5	TAG_KerberosV5_SEC_MECH	**"17"**	KerberosV5

Table B-4 Mechanism Ids

Mechanism Name	**Mechanism Tag**	**Mech Id**	**Base Mech**
CSI_ECMA_Secret	TAG_CSI_ECMA_Secret_SEC_MECH	**“18”**	CSI_ECMA
CSI_ECMA_Hybrid	TAG_CSI_ECMA_Hybrid_SEC_MECH	**“19”**	CSI_ECMA
CSI_ECMA_Public	TAG_CSI_ECMA_Public_SEC_MECH	**“21”**	CSI_ECMA

Cryptographic profile ids are the stringified form of the value of the cryptographic profile constant. For example the id of the cryptographic profile **MD5_RSA** is the string “**20**”. The cryptographic profile ids currently defined are listed below.

Table B-5 Cryptographic Profile Ids

Profile Name	**Profile Id**	**Base Mech**
MD5_RSA	**“20”**	SPKM
MD5_DES_CBC	**“21”**	SPKM
DES_CBC	**“22”**	SPKM
MD5_DES_CBC_SOURCE	**“23”**	SPKM
DES_CBC_SOURCE	**“24”**	SPKM
DES_CBC_DES_MAC	**“10”**	KerberosV5
DES_CBC_MD5	**“11”**	KerberosV5
DES_MAC	**“12”**	KerberosV5
MD5	**“13”**	KerberosV5
FullSecurity	**“1”**	CSI_ECMAS
NoDataConfidentiality	**“2”**	CSI_ECMA
LowGradeConfidentaility	**“3”**	CSI_ECMA
AgreedDefault	**“5”**	CSI_ECMA

A complete mechanism type (used for **MechanismType** parameters) consists of a mechanism id with zero, one or more comma separated cryptographic profiles appended to it. For example the mechanism type “**15,20**” represents **SPKM_1** mechanism with **MD5_RSA** cryptographic profile.

Relationship to Other Services C

C.1 Introduction

This appendix describes the relationship between Object Services and Common Facilities and the security architecture components, if they are to participate in a consistent, secure object system.

C.2 General Relationship to Object Services and Common Facilities

In general, Object Services and Common Facilities, like any application objects, may be unaware of security, and rely on the security enforced automatically on object invocations. As for application objects, access to their operations can be controlled by access policies as described in Section 2.1, “Security Reference Model,” on page 2-1, Section 2.3, “Application Developer’s Interfaces,” on page 2-71, and elsewhere.

An Object Service or Common Facility needs to be aware of security if it needs to enforce security itself. For example, it may need to control access to functions and data at a finer granularity than at object invocation, or need to audit such activities. The way it can do this is described in Section 2.1, “Security Reference Model,” on page 2-1. Existing Object Services should be reviewed to see if such access control and auditing is required.

If an Object Service or Common Facility is required to be part of a more secure system, some assurance of its correct functioning, if security relevant, is needed, even if it is not responsible for enforcing security itself. See Appendix D, “Guidelines for a Trustworthy System” for guidelines on this matter.

Where an Object Service is called by an ORB service as part of object invocation in a secure system, there is a need to ensure security of all the information involved in the invocation. This requires ORB Services to be called in the order required to provide the specified quality of protection. For example, the Transaction Service must be invoked first to obtain the transaction context information before the whole message is protected for integrity and/or confidentiality.

In the following sections, we provide an initial estimation of the relationship between Security Service and other existing services and facilities.

C.3 Relationship with Specific Object Services

C.3.1 Naming Service

For security, the object must be correctly identified wherever it is within the distributed object system. The Naming Service must do this successfully in an environment where an object name is unique within a naming context, and name spaces are federated. (However, to provide the required proof of identity, objects, and/or the gatekeepers which give access to them will be authenticated using a separate Authentication Service.) See Appendix E, Section E.3.2, “Basis of Trust,” on page E-9, for additional information about the relationship between security and names.

C.3.2 Event Service

The implementation of a Security Audit Service may involve the use of Event Service objects for the routing of both audits and alarms.

However, this is only possible if the Event Service itself is secure in that it protects the audit trail from modification and deletion. It must also be able to guard against recursion if it audits its own activities.

C.3.3 Persistent Object Service

No explicit use is made of this service. Audit trails may be saved using this service, in which case the implementation of the Persistent Object Service must ensure that data stored and retrieved through it is not tampered with by unauthorized entities. If it is used in the implementation of Security Service or by a secure application, it must follow the guidelines in Appendix D, “Guidelines for a Trustworthy System.”

C.3.4 Time Service

The Security Service uses the data types for time, timestamps, and time intervals as defined by the Time Service, so that applications can readily use the Time Service defined interfaces to manipulate the time data that the Security Service uses. The interfaces of Security Service do not explicitly pass any interfaces defined in the Time Service.

C.3.5 Other Services

The other services are not used explicitly. If any of them are used in the implementation of Security Service or by a secure application, it must be verified that the service used follows the guidelines in Appendix D.

C.4 Relationship with Common Facilities

Because Management Services have been identified as Common Facilities in the Object Management Architecture, only minimal, security-specific administration interfaces are specified here. When Common Facilities Management services are specified, they will need to take into account the need for security management and administration identified in this specification. Also, such management services will themselves need to be secure.

This specification adds certain basic interfaces to CORBA, which form the basis for the minimal policy administration related interfaces and functionality that has been provided. Future management facilities are expected to build upon this foundation.

Conformance Details and Statement D

D.1 Introduction

CORBA Security Feature Packages include:

- **Main security functionality**. There are two possible levels.
    - ***Level 1***: This provides a first level of security for applications unaware of security, and for those that have limited requirements to enforce their own security in terms of access controls and auditing.
    - ***Level 2***: This provides more security facilities, and allows applications to control the security provided at object invocation. It also includes administration of security policy, allowing applications administering policy to be portable.
- **Security Functionality Options.** These are functions expected to be required in several ORBs, so are worth including in this specification, but are not generally required enough to form part of one of the main security functionality levels previously specified. There is only one such option in the specification.
    - ***Non-Repudiation***: This provides generation and checking of evidence so that actions cannot be repudiated.
- **Security Replaceability**. This specification is designed to allow security policies to be replaced. The additional policies must also conform to this specification. This includes, for example, new Access Polices. Security Replaceability specifies if and how the ORB fits with different security services. There are two possibilities.
    - ***ORB Services replaceability***: The ORB uses interceptor interfaces to call on object services, including security ones. It must use the specified interceptor interfaces and call the interceptors in the specified order. An ORB conforming to this does not include any significant security-specific code, as that is in the interceptors.

- ***Security Service replaceability***: The ORB may or may not use interceptors, but all calls on security services are made via the replaceability interfaces specified in Section 2.5, "Implementor's Security Interfaces," on page 2-143. These interfaces are positioned so that the security services do not need to understand how the ORB works, so they can be replaced independently of that knowledge.

An ORB that supports one or both of these replaceability options is said to be Security Ready (i.e., support no security functionality itself, but be ready to have security added).

Note: Some replaceability of the security mechanism used for secure associations may still be provided if the implementation uses some standard generic interface for security services such as GSS-API.

- **Secure Interoperability using SECIOP**: An ORB supporting this can generate/use security information in the IOR and can send/receive secure requests to/from other ORBs using the GIOP/IIOP protocol with the security (SECIOP) enhancements defined in Section 3.2, "Secure Inter-ORB Protocol (SECIOP)," on page 3-34, providing they can both use the same underlying security mechanism and algorithms for security associations.
- **Common Secure Interoperability (CSI) Feature packages**: These feature packages each provide different levels of secure interoperability. There are three functionality levels for Common Secure Interoperability (CSI).

All levels can be used in distributed secure CORBA compliant object systems where clients and objects may run on different ORBs and different operating systems. At all levels, security functionality supported during an object request includes (mutual) authentication between client and target and protection of messages - for integrity, and when using an appropriate cryptographic profile, also for confidentiality.

An ORB conforming to CSI level 2 can support all the security functionality described in the CORBA Security specification. Facilities are more restricted at levels 0 and 1. The three levels are:

1. ***Identity based policies without delegation (CSI level 0)***: At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target, and this cannot be delegated to further objects. If further objects are called, the identity will be that of the intermediate object, not the initiator of the chain of object calls.

2. ***Identity based policies with unrestricted delegation (CSI level 1)***: At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target. The identity can be delegated to other objects on further object invocations, and there are no restrictions on its delegation, so intermediate objects can impersonate the user. (This is the impersonation form of simple delegation defined in Section 2.1.6.2, "Overview of Delegation Schemes," on page 2-14.)

3. ***Identity & privilege based policies with controlled delegation (CSI level 2)***: At this level, attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions, so the initiating principal can control their use. Optionally, composite delegation is supported, so the attributes of more than one principal can be transmitted. Therefore, it provides interoperability for ORBs conforming to all CORBA Security functionality.

An ORB that interoperates securely must provide at least one of the CSI packages. For the definitive statement on conformance requirements see Appendix C "Conformance Details."

- **Common Security Protocol packages**: The choice of protocol to use depends on the mechanism type required and the facilities required by the range of applications expected to use it. Common Security Protocols define the details of the tokens in the IIOP and SECIOP messages as applicable. Four protocols are defined:

1. ***SPKM Protocol***: This protocol supports identity based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on the definition in [20]. The use of SPKM in CORBA interoperability is based on the SECIOP extensions to IIOP.

2. ***GSS Kerberos Protoco****l*: This protocol supports identity based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0). The GSS Kerberos protocol is based on the [12] which itself is a profile of [13]. The use of Kerberos in CORBA interoperability is based on the SECIOP extensions to IIOP.

3. ***CSI-ECMA protocol***: This protocol supports identity and privilege based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0). For keys assigned to principals, it has the following options:
   - It can use either secret or public key technology.
   - It uses public key technology for keys assigned to trusted authorities.

   The CSI-ECMA protocol is based on the ECMA GSS-API Mechanism as defined in ECMA 235, but is a significant subset of this - the SESAME profile as defined in [16]. It is designed to allow the addition of new mechanism options in the future; some of these are already defined in ECMA 235. The use of CSI-ECMA in CORBA interoperability use the SECIOP extensions to IIOP

DCE-CIOP*:* An ORB supporting this option provides secure interoperability using DCE Security together with the Security extensions to DCE-CIOP.

4. ***SSL protocol***: This protocol supports identity based policies without delegation (CSI level 0). The SSL protocol is based on the definition in [21]. The use of SSL in CORBA interoperability does not depend on the SECIOP extensions to IIOP.

An ORB that interoperates securely must do so using one of these protocol packages. For the definitive statement on conformance requirements see Appendix E “Conformance Statement.”

D.2 Conformance Requirements

An ORB must meet the following requirements to claim conformance to the CORBA Security specification:

- To claim conformance to the **CORBA Security** interfaces it must support the following feature packages:
  - **Security Functionality Level 1**.
- To claim conformance to **CORBA Secure Interoperability** it must support the following feature packages:
  - **Secure Interoperability using SECIOP**.
  - **CSI Level 1**.
  - **GSS Kerberos Protocol using MD5 Cryptographic profile**.
- Conformance to any of the other feature packages may be claimed in addition to the base conformance specified in the previous bullet item, by providing the interfaces, facilities and support for protocols specified in that package, as described further in the following sections.

The conformance statement required for a CORBA Security conformant implementation is defined in Appendix F, “Facilities Not in This Specification.” Appendix F includes two checklists, one for functionality and the other for interoperability, which can be completed to show what the ORB conforms to; they are reproduced next. A main security functionality level must always be specified. Functional Options, Security Replaceability, and Secure Interoperability should be indicated by checking the boxes corresponding to the function supported by the ORB.

Table C-1 CORBA Security Functionality Checklist

Main Functionality		**Functionality Options**	**Security Replaceability**		
Level 1	**Level 2**	**Non Repudiation**	**ORB Services**	**Security Services**	**Security Ready**

Table C-2 CORBA Secure Interoperability Checklist

<table>
<tr><th>Interop</th><th colspan="7">IIOP</th><th>DCE-</th></tr>
<tr><th rowspan="3">Level</th><th colspan="6">SECIOP</th><th rowspan="3">SSL</th><th rowspan="3">CIOP</th></tr>
<tr><th colspan="2">SPKM</th><th rowspan="2">Kerberos</th><th colspan="3">CSI-ECMA</th></tr>
<tr><td>SPKM 1</td><td>SPKM 2</td><td>Private</td><td>Public</td><td>Hybrid</td></tr>
</table>

Table C-2 CORBA Secure Interoperability Checklist

Interop	IIOP							DCE-
Level 0								
Level 1	XXXX	XXXX					XXX	
Level 2	XXXX	XXXX	XXXXX				XXX	

D.3 Security Functionality Level 1

Security Functionality Level 1 provides:

- A level of security functionality available to applications unaware of security. (It will, of course, also provide this functionality to applications aware of security.) This level includes security of the invocation between client and target object, simple delegation of client security attributes to targets, ORB-enforced access control checks, and auditing of security-relevant system events.
- An interface through which a security-aware application can retrieve security attributes, which it may use to enforce its own security policies (e.g., to control access to its own attributes and operations).

D.3.1 Security Functionality Required

An ORB supporting Level 1 security functionality must provide the following security features for all applications, whether they are security-aware or not.

- Allow users and other principals to be authenticated, though this may be done outside the object system.
- Provide security of the invocation between client and target object including:
  - Establishment of trust between them, where needed. At Level 1, this may be supported by ORB level security services or can be achieved in any other secure way. For example, it could use secure lower-layer communications. Mutual authentication need not be supported.
  - Integrity and/or confidentiality of requests and responses between them.
  - Control of whether this client can access this object. At this level, access controls can be based on “sets” of subjects and “sets” of objects. Details of the Access Policy and how this is administered are not specified.
- At an intermediate object in a chain of calls, the ability to be able to either delegate the incoming credentials or use those of the intermediate object itself.
- Auditing of the mandatory set of system’s security-relevant events specified in Appendix A, Consolidated OMG IDL. In some cases, the events to be audited may occur, and be audited, outside the object system (for example, in underlying security services). In this case, the conformance statement must identify the product responsible for generating the record of such an event (or choice of product, for example, when the ORB is portable to different authentication services).

  At this level, auditing of object invocations need not be selectable. However, it must be possible to ensure that certain events are audited (see Appendix B, Section B.11.3, "Audit Event Families and Types," on page B-28, for the list of mandatory events).

Note – For security aware applications, it must also make the privileges of authenticated principals available to applications for use in application access control decisions.

These facilities require the ORB and security services to be initialized correctly. For example, the Current object at the client must be initialized with a reference to a credentials object for the appropriate principal.

D.3.2 Security Interfaces Supported

Security interfaces available to applications may be limited to:

- **get_service_information** providing security options and details (see Section 2.3.2, "Finding Security Features," on page 2-73).
- **get_attributes** on **Current** (see Interfaces under Section 2.3.7, "Security Operations on Current," on page 2-93).

No administrative interfaces are mandatory at this level.

D.3.3 Other Security Conformance

An ORB providing Security Functionality Level 1 may also conform to other security options. For example, it may also:

- Support some of the Security Functionality Options specified in,Section D.5, "Security Functionality Optional Packages," on page D-8.
- Provide security replaceability using either of the replaceability options.
- Provide secure interoperability, though in this case, will need to provide security associations at the ORB level (not lower-layer communications) as the protocol assumes security tokens are at this level.

D.4 Security Functionality Level 2

This is the functionality level that supports most of the application interfaces defined in Section 2.3, "Application Developer's Interfaces," on page 2-71, and the administrative interfaces defined in Section 2.4, "Administrator's Interfaces," on page 2-116. It provides a competitive level of security functionality for most situations.

D.4.1 Security Functionality Required

An ORB that supports Security Functionality Level 2 supports the functionality in Security Level 1 previously defined, and also:

- Principals can be authenticated outside or inside the object system.
- Security of the invocation between client and target objects is enhanced.
  - Establishment of trust and message protection can be done at the ORB level, so security below this (for example, in the lower layer communications) is not required (though may be used for some functions).
  - Further integrity options can be requested (e.g., replay protection and detection of messages out of sequence) but need not be supported.
  - The standard **DomainAccessPolicy** is supported for control of access to operations on objects.
  - Selective auditing of methods on objects is supported.
- Applications can control the options used on secure invocations. It can:
  - Choose the quality of protection of messages required (subject to policy controls).
  - Change the privileges in credentials.
  - Choose which credentials are to be used for object invocation.
  - Specify whether these can just be used at the target (e.g. for access control) or whether they can also be delegated to further objects.
- No further delegation facilities are mandatory, but the application can request “composite” delegation, and the target can obtain all credentials passed, in systems that support this. Note that “composite” here just specifies that both received credentials and the intermediate’s own credentials should be used. It does not specify whether this is done by combining the credentials or linking them.
- Administrators can specify security policies using domain managers and policy objects as specified in Section 2.4, “Administrator’s Interfaces,” on page 2-116. The security policy types supported at Level 2 are all those defined in Section 2.4, “Administrator’s Interfaces,” on page 2-116 except non-repudiation. The standard policy management interfaces for each of the Level 2 policies is supported.
- Applications can find out what security policies apply to them. This includes policies they enforce themselves (e.g., which events types to audit) and some policies the ORB enforces for them (e.g., default qop, delegation mode).
- ORBs (and ORB Services, if supported) can find out what security policies apply to them. They can then use these policy objects to make decisions about what security is needed (check if access is permitted, check if auditing is required) or get the information needed to enforce policy (get QOP, delegation mode, etc.) depending on policy type.

As at Level 1, these facilities require the ORB and security services to be initialized correctly.

D.4.2 Security Interfaces Supported

Interfaces supported at this level are:

- All application interfaces defined in Section 2.3, “Application Developer’s Interfaces,” on page 2-71, except those in Section 2.1.7, “Non-repudiation,” on page 2-18.

- All security policy administration interfaces defined in Section 15.6, Administrator's Interfaces (except those for the non-repudiation policy).

Note that some of these interfaces may raise a **CORBA::NO-IMPLEMENT** exception, as not ORBs conforming to Level 2 Security need implement all possible values of all parameters. This will happen when:

- A privilege attribute is requested of a type that is not supported (attribute types supported are defined in Appendix B, Section B.2, "General Security Data Module," on page B-1).
- A delegation mode is requested, which is not supported.
- A communication direction for association options is requested, which is not supported.

D.4.3 Other Security Conformance

An ORB providing Security Functionality Level 2 may also conform to other security options. For example, it may also:

- Support some of the Security Functionality Options specified in Section D.6, "Security Replaceability," on page D-9.
- Provide security replaceability, using either of the replaceability options.
- Provide secure interoperability.

D.5 Security Functionality Optional Packages

An ORB may also conform to optional security functionality defined in this specification. Only one optional facilities is specified: non-repudiation.

Also, some requirements on conformance of additional facilities are specified.

D.5.1 Non-repudiation

D.5.1.1 Security Functionality

An ORB conforming to this must support the non-repudiation facilities for generating and verifying evidence described in Section 2.2.5.1, "The Model as Seen by Applications," on page 2-41. Note that these use **NRCredentials**, the attributes in which may be the same as in the credentials used for other security facilities. Where non-repudiation is supported, the credentials acquired from the environment or generated by the authenticate operation must be able to support non-repudiation.

D.5.1.2 Security Operations Supported

The following operations must be supported. All are available to applications. They are:

- **set_/get_NR_features** as defined in Section 2.1.7, "Non-repudiation," on page 2-18.
- **generate_token**, **verify_evidence**, **form_complete_evidence** and **get_token_details** of **NRCredentials** object as defined in Section 2.1.7, "Non-repudiation," on page 2-18.
- Use of **set/get_credentials** on **Current** specifying the type of credentials to be used is **NRCredentials**.
- **NRPolicy** object with associated interfaces as in Section 2.1.7, "Non-repudiation," on page 2-18.

D.5.1.3 Fit with Other Security Conformance

Non-repudiation requires use of credentials; thus it can only be used with ORBs, which support some of the interfaces defined in Security Functionality level 2. However, conformance to all of Security Functionality Level 2 is not a prerequisite for conformance to the non-repudiation security functionality option.

Secure interoperability as defined in Section D.7, "Secure Interoperability," on page D-11, is not affected by non-repudiation. The evidence may be passed on an invocation as a parameter to a request, but the ORB need not be aware of this.

The current specification does not specify interoperability of evidence (i.e. one non-repudiation service handling evidence generated by another).

D.5.2 Conformance of Additional Policies

This specification is designed to allow security policies to be replaced. The additional policies must also conform to some of the interfaces in this specification if they are used to replace the standard policies automatically enforced on object invocation.

The case described next is for the addition of a new Access Policy which can be used for controlling access to objects automatically, replacing the standard **DomainAccessPolicy**.

Clearly, other policies can be replaced. For example, the audit policy could be replaced by one that used different selectors, or the delegation policy could be replaced by one that supported more advanced features.

D.6 Security Replaceability

This specifies how an ORB can fit with security services, which may not come from the same vendor as the ORB. As explained above, there are two levels where this can be done (apart from any underlying APIs used by an implementation).

D.6.1 Security Features Replaceability

Conformance to this allows security features to be replaced.

If it is provided without conformance to the ORB Service replaceability option (see Section D.6.2, “ORB Services Replaceability,” on page D-10), it requires the ORB to have a reasonable understanding of security, handling credentials, etc. and knowing when and how to call on the right security services.

Support for this replaceability option requires an ORB (or the ORB Services it uses) to use the implementation-level security interfaces as defined in Section 2.5, “Implementor’s Security Interfaces,” on page 2-143. This includes:

- The **Vault**, **Security Context**, **Access Decision**, **Audit** and **Principal Authentication** objects defined in Section 2.5.2, “Implementation-Level Security Object Interfaces,” on page 2-149.
- Certain features of the CORBA Core needed for ORB Service Replaceability can be found in the *Common Object Request Broker: Architecture and Specification*.

D.6.2 ORB Services Replaceability

Conformance to this allows an ORB to know little about security except which interceptors to call in what order. This is intended for ORBs, which may use different ORB services from different vendors, and require these to fit together. It therefore provides a generic way of calling a variety of ORB Services, not just security ones. It also assumes that any of these services may have associated policies, which control some of their actions.

Support for this replaceability option requires an ORB to:

- Use the Interceptor interfaces defined in the Interceptor chapter of the *Common Object Request Broker: Architecture and Specification* to call security interceptors defined in Section 2.5.1, “Security Interceptors,” on page 2-144, in the order defined there.
- Use the **get_policy** operation (and the associated security policy operations such as **access_allowed**, **audit_needed** defined in Section 2.1.4, “Access Control Model,” on page 2-7 and Section 2.3.8, “Security Audit,” on page 2-100 respectively, for access control and audit and also **get_association_options** and **get_delegation_mode** defined in Section 2.4.6, “Secure Invocation and Delegation Policies,” on page 2-135, for association options, quality of protection of messages, and delegation).

D.6.3 Security Ready for Replaceability

An ORB is Security Ready for Replaceability if it does not provide any security functionality itself, but does support one of the security replaceability options.

D.6.3.1 Security Functionality Required

An ORB that is Security Ready does not have to provide any security functionality, though must correctly respond to a request for the security features supported.

D.6.3.2 Security Interfaces Supported

- **get_service_information** operation providing security options and details (see Section 2.3.2, “Finding Security Features,” on page 2-73).
- **get_current** operation to obtain the Current object for the execution context (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*).

D.6.3.3 Other Security Conformance

An ORB that is Security Ready for replaceability supports one of the replaceability options. This should be done in such a way that the ORB can work without security, but can take advantage of security services when they become available. So it calls on the replaceability interfaces correctly (using dummy routines to replace security services when these are needed, but not available).

D.7 Secure Interoperability

The definition of secure interoperability in this document specifies that a conformant ORB can:

- Generate, and take appropriate action on, Interoperable Object References (IORs), which include security tags as specified in Section 3.1.4, “CORBA Interoperable Object Reference with Security,” on page 3-7.
- Transmit and receive the security tokens needed to establish security associations, and also the protected messages used for protected requests and responses once the association has been established according to the protocol defined in Section 3.2, “Secure Inter-ORB Protocol (SECIOP),” on page 3-34

Note that a Security Ready ORB (i.e., with no built-in security functionality) may, by additions of appropriate security services, conform to secure interoperability.

For ORBs to interoperate securely, they must choose to use the same mechanism, algorithms, etc. (or use a bridge between them, if available). A set of standard security mechanisms and algorithms are described in subsections.

D.7.1 Standard Secure Interoperability

An ORB that conforms to this must support the security-enhanced IOR defined in Section 3.1.4, “CORBA Interoperable Object Reference with Security,” on page 3-7, and also GIOP/IIOP protocol with the SECIOP enhancements as defined in Section 3.2, “Secure Inter-ORB Protocol (SECIOP),” on page 3-34.

As for CORBA 2, this may be done by immediate bridges or half bridges. (However, use of half bridges implies more complex trust relationships, which some systems may not be able to support.) This allows a large range of security mechanisms to be used.

D.7.2 Common Secure Interoperability Levels

There are three functionality levels for Common Secure Interoperability (CSI). An example of the difference in use of the three levels is explained in Section D.7.2, “Common Secure Interoperability Levels,” on page D-12.

All levels can be used in distributed secure CORBA compliant object systems where clients and objects may run on different ORBs and different operating systems. At all levels, security functionality supported during an object request includes (mutual) authentication between client and target and protection of messages - for integrity, and when using an appropriate cryptographic profile, also for confidentiality.

An ORB conforming to CSI level 2 can support all the security functionality described in this specification. Facilities that are supportable at levels 0 and 1 are more restricted. The three levels are:

1. Identity based policies without delegation (CSI level 0)

At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target, and this cannot be delegated to further objects. If further objects are called, the identity will be that of the intermediate object, not the initiator of the chain of object calls.

Access and audit policies at this level are based on the identity of the immediate invoker. So access and audit policies in encapsulated objects which depend on the initiator of the chain, can only be used at the point of entry to the object system, not in further objects encapsulated by it.

As the attributes of principals are not delegated, environments should not be trusted to pass on principal information which should be controlled.

Examples of applications which can use level 0 facilities are wrapped legacy applications and telephone switches. If a CSI level 0 ORB also supports non-repudiation, it can also be used for other types of applications such as electronic funds transfer.

2. Identity based policies with unrestricted delegation (CSI level 1)

At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target. The identity can be delegated to other objects on further object invocations, and there are no restrictions on its delegation, so intermediate objects can impersonate the user. (This is the impersonation form of simple delegation defined in Section 2.1.6, “Delegation,” on page 2-13.)

Access and audit policies at this level can be based on the identity of the initiating principal or immediate invoker, depending on the delegation policy.

As delegation is not restricted, once an initiator has delegated his identity, it must trust the objects it calls not to abuse its delegated rights to act as the initiator. In practice, this will limit the type of environment in which level 1 should be used to relatively closed environments.

An example of an application environment which can use level 1 facilities is a back office system protected by firewalls where identity based policies are acceptable.

3. Identity & privilege based policies with controlled delegation (CSI level 2)

At this level, attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions, so the initiating principal can control their use. Optionally, composite delegation is supported, so the attributes of more than one principal can be transmitted. Therefore, it provides interoperability for ORBs conforming to all CORBA Security functionality.

Access and audit policies are based on the attributes of initiating principals. At this level, a wider range of policies can be supported (e.g., role based access controls and mandatory access controls using the initiating principal's security clearance).

At this level, an initiator needs to trust those targets which it has allowed to use its attributes not to abuse these. It does not have to trust these targets not to delegate the attributes outside the trusted set of targets, as the delegation controls can be used to prevent this.

This level can be used for a wide range of applications in large enterprise and inter-enterprise networks.

D.7.3 SECIOP Hosted Interoperability Mechanisms

The following conformance can be claimed:

- SPKM at level 0 by providing the specified CSI level using the SPKM protocol (mechanism **SPKM_1** and optionally also **SPKM_2**).
- **KerberosV5** at level 0 or 1 by providing the specified CSI level using the Kerberos protocol.
- CSI-ECMA Public Key at level 0, 1, or 2 by providing the specified level of CSI functionality using the CSI-ECMA protocol with the public key option (mechanism **CSI_ECMA_Public**).
- CSI-ECMA Secret Key at level 0, 1, or 2 by providing the specified CSI level using the CSI-ECMA protocol with the secret key option (mechanism **CSI_ECMA_Secret**).
- CSI-ECMA Hybrid at level 0, 1, or 2 by providing the specified CSI level using the CSI-ECMA protocol with the hybrid key option (mechanism **CSI_ECMA_Hybrid**).

In addition, a conformant ORB must specify all the cryptographic profiles it supports.

D.7.4 Secure Interoperability with SSL

Conformance can be claimed for CORBA Security based on SSL by providing CSI level 0 functionality using SSL on IIOP using any of the cryptographic profiles defined in[21]. A conformant ORB must specify which of the cryptographic profiles are supported by it.

D.7.5 Secure Interoperability with DCE-CIOP

An ORB that conforms to this must conform to Standard Secure Interoperability using GIOP/IIOP as described in Section D.7.1, "Standard Secure Interoperability," on page D-11, and also support secure interoperability using DCE-CIOP as defined in Section 3.8, "DCE-CIOP with Security," on page 3-105.

Both the Kerberos V5 based SECIOP Security and DCE Security must be supported for this option. Any version of DCE up to and including DCE 1.1 is supported; the DCE interfaces and protocols are specified in [5]

D.8 Conformance Statement

D.8.1 Introduction

A secure object system, like any secure system, should not only provide security functionality, but should also provide some assurance of the correctness and effectiveness of that functionality.

Each OMG-compliant secure or security ready implementation must therefore include in its documentation a conformance statement describing:

- The product's supported security functionality levels and options, security replaceability, and security interoperability, as described in Appendix C.
- The vendor's assurance argument that demonstrates how effectively the product provides its specified security functionality and security policies.
- Constraints on the use of the product to ensure security conformance.

The vendor provides the conformance statement so that a potential product user can make an informed decision on whether a product is appropriate for a particular application. Ordinary descriptive documentation is not required as part of an OMG-compliant product. However, because the CORBA security specification provides a general security framework rather than a single model, there are many different kinds of secure ORB implementations that conform to the framework. For example, some systems may have greater flexibility and support customized security policies, while other systems may come with a single built-in policy. Some systems may strive for a high level of security assurance, while others provide minimal assurance. The conformance statement will help the user understand the security features provided by the product.

Some products will undergo an independent formal security evaluation (such as ones meeting the ITSEC or TCSEC). The OMG security conformance statement does not take the place of a formal evaluation, but may refer to formal assurance documentation, if it exists. When formal evaluations are not required (often the case in commercial systems), it is expected that the product's security conformance statement along with supporting product documentation will provide an adequate description of security functionality and assurance.

D.8.2 Conformance Template Overview

The following template specifies the contents for CORBA security conformance statements. Guidelines for using this template are provided in Section, Conformance Guidelines.

CORBA Security Conformance Statement

<date>

<product identification>

<vendor identification>

1. Introduction

1.1 Summary of Security Conformance

1.2 Scope of Product

1.3 Security Overview

2. Security Conformance

2.1 Main Security Functionality Level

2.2 Security Functionality Options

2.3 Security Replaceability

2.4 Secure Interoperability

3. Assurance

3.1 Philosophy of Protection

3.2 Threats

3.3 Security Policies

3.4 Security Protection Mechanisms

3.5 Environmental Support

3.6 Configuration Constraints

3.7 Security Policy Extensions

4. Supplemental Product Information

D.8.3 Conformance Guidelines

The guidelines in this section are intended to help the ORB implementor determine which information belongs in each section of the conformance statement. The statement will often be accompanied by product documentation to provide some of the information needed.

1. Introduction

1.1 Summary of Security Conformance

This section should give a summary of the security conformance provided by the product. The summary is in the form of a table with boxes that are ticked to show the relevant conformance.

Table 15-3 CORBA Security Functionality Checklist

Main Functionality		**Functionality Options**	**Security Replaceability**		
Level 1	**Level 2**	**Non Repudiation**	**ORB Services**	**Security Services**	**Security Ready**

Table 15-4 CORBA Secure Interoperability Checklist

Interop	**IIOP**							**DCE-**
Level	**SECIOP**						**SSL**	**CIOP**
	SPKM		**Kerberos**	**CSI-ECMA**				
	SPKM 1	SPKM 2		Privat e	Public	Hybri d		
Level 0								
Level 1	**XXXX**	**XXXX**					**XXX**	
Level 2	**XXXX**	**XXXX**	**XXXXX**				**XXX**	

For the main security functionality level, one of the boxes must be selected (either Level 1 or Level 2), though note that an ORB can be just Security Ready, so does not support either of the main security functionality levels. For security functionality options, security replaceability, and secure interoperability, the appropriate boxes should be selected.

1.2 Scope of Product

This section should define what security components this product offers. Examples are:

- ORB plus all security services needed to support it plus other object services fitting with it and meeting the assurance criteria.
- Security-ready ORB.
- Security Services, which can be used with a security-ready ORB.

1.3 Security Overview

This section should give an overview of the product's security features.

2. Security Conformance

2.1 Main Security Functionality Level

This section should define which main security functionality level this product supports, Level 1 or Level 2.

This should also include any qualifications on that support. For example, any interpretation of the CORBA security specification and how it is supported, any bells and whistles around the published interfaces, and any limitations on support for this level.

As in the conformance level descriptions, the description should be divided into:

- The security functionality provided by the product
- The application developer's interfaces
- The administrative interfaces

2.2 Security Functionality Options

This section should define which functionality options are provided, in particular the support for non-repudiation.

For non-repudiation, as this is a published interface in this specification, it should be accompanied by a qualification statement if needed, as for the main security functionality level.

2.3 Security Replaceability

This section should define whether the product supports replaceability of security services, ORB services, or neither.

This should also include any qualifications on that support. For example, any interpretation of the CORBA security specification and how it is supported, any bells and whistles around the published interfaces, and any limitations on support for this conformance option.

2.4 Secure Interoperability

This section should define whether the product supports SECIOP based secure interoperability, DCE-CIOP based interoperability, SSL based interoperability, or none. As with the previous sections, qualifications of the support, interpretations of the CORBA specification, and limitations should be included as needed.

2.5 Level of Interoperability

This section should specify what level of interoperability is supported by the ORB. As with the previous sections, qualifications of the support, interpretations of the CORBA specification, and limitations should be included as needed.

2.6 Mechanism Profiles

This section should specify what mechanism and cryptographic profiles for interoperability are supported by the ORB. As with the previous sections, qualifications of the support, interpretations of the CORBA specification, and limitations should be included as needed.

3. Assurance

If the product already has supporting assurance documentation (for example, because it is being formally evaluated), much of this section may be satisfied by references to such documentation. Appendix E, Guidelines for a Trustworthy System, provides general discussions of many of the topics described here, particularly the basis of trust needed for each of the architecture object models.

3.1 Philosophy of Protection

Overview of supported security policies, security mechanisms and supporting mechanisms.

3.2 Threats

Description of specific threats intended to be addressed by the system security policy, as well as those not addressed.

3.3 Security Policies

Description of any predefined policies, including

- Classes of entities (such as clients, objects) controlled by security policy
- Modes of access (conditions that allow active entities to access objects)
- Use of domains (policy, trust, technology)
- Requirements for authentication of principal, client and target objects
- Requirements for trusted path between principals, clients, ORBs, and target objects
- Delegation model
- Security of communications
- Accountability requirements (audit, non-repudiation)

- Environmental assumptions of the policy (e.g. classes of users, LAN/WAN, physical protection)

3.4 Security Protection Mechanisms

- Rationale for approach
- Identification of components, which must function properly for security policies to be enforced
- Description of mechanisms used to enforce security policy
- How protection mechanisms are distributed in the architecture
- Why security mechanisms (such as access control) are always invoked and tamper-proof

3.5 Environmental Support

- How the underlying environment (such as operating systems, generation tools, hardware, network services, time services, security technology) are used in providing assurance
- How installation tools ensure secure configuration
- How security management and administration maintains secure configuration

3.6 Configuration Constraints

Constraints to ensure that system security assurance is preserved, for example:

- Requirements on use and development of: clients, target objects, legacy software
- Limitations on interoperability
- Required software and hardware configuration

3.7 Security Policy Extensions

- Supported security policy extensions, if applicable
- Limitations of extensions
- Requirements imposed on developers to ensure trustworthiness of policy extensions
- Supported interactions and compositions of security policies

4. Supplemental Product Information

Supplemental product information is included at the vendor’s discretion. It can be used to describe, for example:

- Additional security features, not covered by the CORBA Security specification
- The impact of security mechanisms on existing applications

Guidelines for a Trustworthy System E

E.1 Introduction

This appendix provides some general guidelines for helping ORB implementors produce a trustworthy system. The intention is to have all information related to trustworthiness and assurance in this appendix, to explain how the specification has taken into account the requirements for assurance, and also to show how conformant implementations can have different levels of assurance.

The remainder of the introduction first provides the rationale for including these guidelines in the specification, and then gives some background on trustworthiness and assurance. Section E.2, “Protecting Against Threats,” on page E-3, describes the threats and countermeasures relevant to a CORBA security implementation. Section E.3 through E.6 provide the architecture and implementation guidelines for each security object model described in Section 2.2, “Security Architecture,” on page 2-28.

E.1.1 Purpose of Guidelines

The security standards proposed in this specification have been deliberately chosen to allow flexibility in the security features, which can be provided. The specification can support significantly different security policies and mechanisms for security functions such as access control, audit and authentication. However, there is an overall security model which applies whatever the security policy. This is described in the earlier sections of the document.

There is also flexibility in the level of security assurance, which can be provided, conforming to this model and these standards. This appendix describes the trustworthiness issues underlying the security model and interfaces described earlier in the document, and provides implementation guidance on what components of the architecture need to be trusted and why. Note that trust requirements assume conformance to all of the security models, including the implementor’s view, as the implementation affects trustworthiness. If a CORBA security implementation conforms to the security features replaceability level, but not the ORB services one, any requirements on ORB services will apply to the ORB. Trustworthiness will also depend on several other implementation choices, such as the particular security technology used.

E.1.2 Trustworthiness

Before an enterprise places valuable business assets within an IT system, enterprise management must decide whether the assets will be adequately protected by the system. Management must be convinced that the particular system configuration is sufficiently *trustworthy* to meet the security needs of the enterprise environment. Security trustworthiness is thus the ability of a system to protect resources from exposure to misuse through malicious or inadvertent means.

The basis for trust in distributed systems differs from host-centric stand-alone systems largely for two reasons. First, the assignment of trust in a distributed system is not isolated to a single global system mechanism. Second, the degree of trust in elements of distributed systems (particularly distributed *object* systems) may change dynamically over time, whereas in host-centric systems trustworthiness is typically static. In many cases, trust in distributed systems must be seen in the context of mutual suspicion.

E.1.3 Assurance

Assurance is a qualitative measure of trustworthiness; assurance is the confidence that a system meets enterprise security needs. The qualitative nature of assurance means that enterprises may have different assurance guidelines for an equivalent level of confidence in security. Some organizations may need extensive evaluation criteria, while other organizations need very little evidence of trustworthiness.

It is necessary to set a context by which CORBA developers and end-users of the CORBA Security specification may evaluate the level of security to meet their needs. A single overall trust model that underlies the security reference model and architecture (as described elsewhere in this specification) can set this context for closed systems, but it is unlikely that a single trust model exists for the diversity of open distributed systems likely to populate the distributed object technology world.

To support a balanced approach, assurance arguments should be assembled from a set of system building blocks. Concepts of system composition and integration should allow the assurance analysis to be tailored to specific user requirements. Assurance evidence should be carefully packaged to best support enterprise decision-makers during the security trade-off process.

The security object models defined by the CORBA Security specification are the basis for the necessary building blocks. The trust guidelines described in “Guidelines for Structural Model” on page E-8, provide constraints on how these components may relate.

The relationship between assurance and security provides the foundation for the overall security model. The key characteristic is balance. Balanced assurance promotes the use of assurance arguments and evidence appropriate to the level of risk in the system components.

Basic system building blocks, such as those in the CORBA Security specification previously noted, are critical to developing balanced assurance. For example, confidentiality is of most importance to a classified intelligence or military system, whereas data integrity may be of more importance in a computer patient record system. The former relies on assurance in the underlying operating system, where the latter focuses security in application software.

E.2 Protecting Against Threats

An enterprise needs to protect its assets against perceived threats using appropriate security measures. This document addresses security in distributed object systems, so focuses on the threats to assets, software, and data, in such systems.

An enterprise may want to assess the risk of a security breach occurring, against the damage which will be done if it does occur. The enterprise can then decide the best trade-off between the cost of providing protection from such threats and any performance degradation this causes, against the probability of loss of assets. This specification allows options in how security is provided to counter the threats. However, it is expected that many enterprises will not undertake a formal risk assessment, but rely on a standard level of protection for most of their assets, as identified by industry or government criteria. This section describes CORBA-specific security goals, the main distributed system threats, and protection against them. The discussion does not emphasize generic issues of threats and countermeasures, but instead concentrates on issues that are unique to the CORBA security architecture.

E.2.1 Goals of CORBA Security

The overall goals of the CORBA security architecture were described in Section 1.1, “Introduction to Security,” on page 1-2. CORBA security is based on the four fundamental objectives of any secure system:

- Maintain confidentiality of data and/or system resources.
- Preserve data and/or system integrity.
- Maintain accountability.
- Assure data/system availability.

Many of the goals described in Section 1.1, “Introduction to Security,” on page 1-2 are relevant to any IT system that is targeted at large-scale applications. However, some security goals described are specific to the CORBA security architecture. These goals deserve special attention because they surface potential threats that may not be encountered in typical architectures. CORBA-specific security goals include:

- Providing security across a heterogeneous system where different vendors may supply different ORBs.

- Providing purely object-oriented security interfaces.
- Using encapsulation to promote system integrity and to hide the complexity of security mechanisms under simple interfaces.
- Allowing polymorphic implementations of objects based on different underlying mechanisms.
- Ensuring object invocations are protected as required by the security policy.
- Ensuring that the required access control and auditing is performed on object invocation.

The discussion of the architecture and implementation guidelines in Section E.3, “Guidelines for Structural Model,” on page E-8, addresses the mechanisms used to ensure these CORBA-specific security goals, as well as many other generic security issues.

E.2.2 Threats

The CORBA security model needs to take into account all potential threats to a distributed object system. It must be possible to set a security policy and choose security services and mechanisms that can protect against the threats to the level required by a particular enterprise.

A security *threat* is a potential system misuse that could lead to a failure in achieving the system security goals previously described. Section 1.1, “Introduction to Security,” on page 1-2, provided an overview of security threats in a distributed object system. These threats and related attacks include:

- **Information compromise** - the deliberate or accidental disclosure of confidential data (e.g., masquerading, spoofing, eavesdropping).
- **Integrity violations** - the malicious or inadvertent modification or destruction of data or system resources (e.g., trapdoor, virus).
- **Denial of service** - the curtailment or removal of system resources from authorized users (e.g., network flooding).
- **Repudiation of some action** - failure to verify the actual identity of an authorized user and to provide a method for recording the fact (e.g., audit modification).
- **Malicious or inadvertent misuse** - active or passive bypassing of controls by either authorized or unauthorized users (e.g., browsing, inference, harassment).

The threats described above give rise to a wide variety of attacks. Most if not all the threats that pertain to host-centric systems are pertinent to distributed systems. Furthermore, it appears likely that the wide distribution of resources and mediation in truly distributed systems will not only exacerbate the strain on host-centric security services and mechanisms in use today on client/server systems, but also engender new forms of threat.

Threats may be of different strengths. For example, accidental misuse of a system is easier to protect against than malicious attacks by a skilled hacker. This specification does not attempt to counter all threats to a distributed system. Those that should be countered by measures outside the scope of this specification include:

- Denial of service, which may be caused by flooding the communications with traffic. It is assumed that the underlying communications software deals with this threat.
- Traffic analysis.
- Inclusion of rogue code in the system, which gives access to sensitive information. (This affects the build and change control process.)

E.2.3 Vulnerabilities of Distributed Object-Oriented Systems

Vulnerabilities are system weaknesses that leave the system open to one or more of the threats previously described. Information systems are subject to a wide range of vulnerabilities, a number of which are compounded in distributed systems. These vulnerabilities often result from deliberate or unintentional trade-offs made in system design and implementation, usually to achieve other more desirable goals such as increased performance or additional functionality.

Classes of vulnerabilities include:

- An authorized user of the system gaining access to some information which should be hidden from that user, but has not been properly protected (e.g., access controls have not been properly set up or the store occupied by one object has not been cleared out when another reuses the space).
- A user masquerading as someone else, and so obtaining access to whatever that user is authorized to do, resulting in actions being attributed to the wrong person. In a distributed system, a user may delegate his rights to other objects, so they can act on his behalf. This adds the threat of rights being delegated too widely, again, causing a threat of unauthorized access.
- Controls that enforce security being bypassed.
- Eavesdropping on a communication line giving access to confidential data.
- Tampering with communication between objects: modifying, inserting, and deleting items.
- Lack of accountability due, for example, to inadequate identification of users.

System data as well as business data must be protected. For example:

- If a principal’s credentials are successfully obtained by an unauthorized user, they could be used to masquerade as that principal.
- If the security sensitive information in the security context between client and target object is available to an unauthorized user, confidential messages can be read, and it may be possible to modify and resend integrity-protected messages or send false messages without this being detected.

As described earlier, system threats and vulnerabilities are compounded by the complexities of distributed object-based systems. Some of the inherent characteristics of distributed object systems that make them particularly vulnerable include:

- **Dynamic Systems** -- Distributed object systems are always changing. New components are constantly being added, deleted, and modified. Security policies also may be dynamically modified as enterprises change. Dynamic systems are inherently complex, and thus security may be difficult to ensure. For example, in a large distributed object system it will be difficult to update a security policy atomically. While an administrator installs a new policy on some parts of the system, other parts of the system still may be using the old version of the policy. These potential inconsistencies in policy enforcement could lead to a security failure.
- **Mutual Suspicion** -- In a large distributed system, some system components will not trust others. Mistrust could occur at many layers within the architecture: principals, objects, administrators, ORBs, and operating systems may all have varying degrees of trustworthiness. In this environment, there is always the potential to inadvertently place unjustified trust in some system component, thus exposing a vulnerability. Although there are many mechanisms (e.g., cryptographic authentication) to ensure the identity of a remote component, the system security architecture must be carefully structured to ensure that these checks are always performed.
- **Multiple Policy Domains** -- Distributed object systems that interconnect many enterprises are likely to require many different security policy domains, each one enforcing the security requirements of its organization. There is no single security policy and enforcement mechanism that is appropriate for all businesses. As a result, security policies must be able to address interactions across policy domain boundaries. Defining the appropriate policies to enforce across domains may be a difficult job. Mismatched policies could lead to vulnerabilities.
- **Layering of Security Mechanisms** -- Distributed object systems are highly layered, and the security mechanisms for those systems will be layered as well. Complex, potentially nondeterministic interactions at the boundary of the layers is another area for vulnerabilities to occur. A hardware error, for example, could cause security checking code in the ORB to be bypassed, thus violating the policy. The complexity of the layering is further compounded in systems where security enforcement is widely distributed; that is, there is no clear security perimeter containing only a small amount of simple functionality.
- **Complex Administration** -- Finally, large geographically distributed object systems may be difficult to administer. Security administration requires the cooperation of all the administrators, who even may be mutually suspicious. All of the issues listed above lead to complex, error-prone administration. An innocent change to a principal's access rights, for example, could expose a serious vulnerability.

E.2.4 Countermeasures

Some threats are common across most distributed secure systems, so should be countered by standard security features of any OMA-compliant secure systems. However, the level of protection against these threats may vary. Complete protection is almost impossible to achieve. Most enterprises will want a balance between a level of protection against threats which are important to them, and the cost in performance and use of other resources of providing that level of protection.

A number of measures exist for countering or mitigating the effects of the above threats/attacks. Countering these threats requires the use of the security object models described in this specification. Relevant features of the object models include the following:

- Authentication of principals proves who they are, so it is possible to check what they should be able to do. This check can be performed at both client and target object, as the client principal's credentials can be passed to the server.
- Authentication between clients and target objects allows them to check that they are communicating with the right entities.
- Security associations can protect the integrity of the security information in transit between client and target object (e.g., credentials, keys) to prevent theft and replay, and keep the keys used for protecting business data confidential.
- Business data can be integrity-protected in transit so any tampering is detected using the message protection ORB services. (This includes detecting extra or missing messages, and messages out of sequence.)
- Unauthorized access to objects is protected using access controls.
- Misuse of the system can be detected using auditing.
- Segregating (groups of) applications from each other and security services from applications can prevent unauthorized access between them.
- Bypassing of security controls is deterred by use of a Trusted Computing Base (TCB), where security is automatically enforced during object invocation.

Assurance arguments and evidence are frequently founded on the concept of a TCB, which mediates security by segregating the security-relevant functions into a security kernel or reference monitor.

A traditional monolithic TCB approach is not suitable for the open, multiuser, multiple environment situations in which most CORBA users reside. In many cases, for example, secure interoperability of CORBA applications and ORBs may be based on mutual suspicion. TCB scalability issues also argue against typical TCB approaches. Given the complexity of distributed systems, it is not clear whether centralized access mediation is possible in the presence of distributed data and program logic.

Traditional TCB approaches also do not adequately address application security requirements, particularly for many commercial applications. Applications common to the CORBA world such as general purpose DBMSs, financial accounting, electronic commerce, or horizontal common facilities will have many security requirements in addition to those that can be enforced by a central underlying TCB.

Despite the limitations of the traditional TCB, we use the concept of a *distributed TCB* in the assurance discussions of the next section. The concept of a distributed TCB is the collection of objects and mechanisms that must be trusted so that end-to-end security between client and target object is maintained. However, note that depending on the assurance requirements of a particular CORBA security architecture, sensitive data may still be handled by “entrusted” ORB code. Thus, our informal use of the distributed TCB concept may not correspond to other existing models for network TCBs, particularly for minimal assurance commercial CORBA security applications.

E.3 Guidelines for Structural Model

This section provides architecture and implementation guidelines for the structural model of the CORBA security architecture described in Section 2.2, “Security Architecture,” on page 2-28. The security functions provided in the model and the basis for trust are described.

E.3.1 Security Functions

Figure E-1 outlines interactions during a normal use of the system. It gives a simple case, where the application is unaware of security except for calling a security service such as audit. The security interactions include those seen by application objects and secure object system implementors.

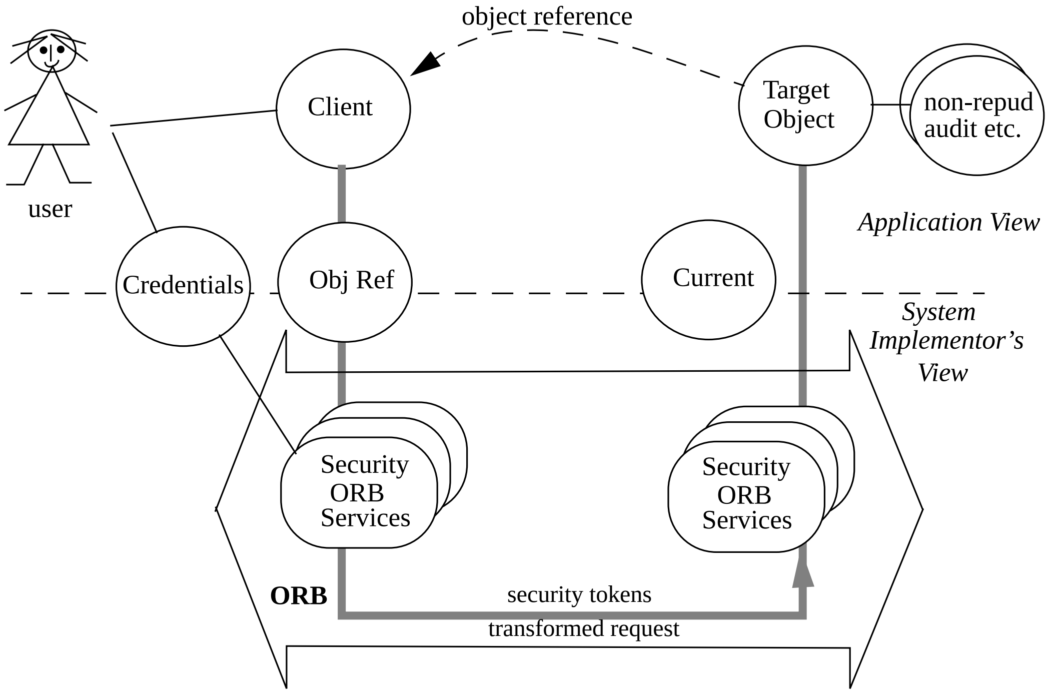


Figure E-1 Normal System Interactions

This diagram is the basis for the discussions of security functions in each of the security object models described next.

E.3.2 Basis of Trust

Enterprise management is responsible for setting the overall security policies and ensuring system enforcement of the policies.

The system developer and systems integrators must provide a system that supports the required level of assurance in the core security functionality. Generally application developers cannot be expected to be aware of all the threats to which the system will be subject, and to put the right countermeasures in place.

Higher levels of security may require the code enforcing it to be formally evaluated according to security criteria such as those of the US TCSEC or European ITSEC.

Distributed Trusted Computing Base

The key security functionality in the system is enforced transparently to the application objects so that it can be provided for application objects, which are security unaware. This key functionality is contained in the distributed TCB of the system. It is therefore responsible for ensuring that:

- Users cannot invoke objects unless they have been authenticated (unless the security policy supports unauthenticated, guest access for some services).
- Security policies on access control, audit, and security association are enforced on object invocation. This includes policies for message protection, both confidentiality (ensuring confidential data cannot be read) and integrity (ensuring any corruption of data in transit is detected).
- A principal’s credentials are automatically transferred on object invocation if required, so the access control and other security policies can be enforced at the server object.
- Application objects which do not trust each other cannot interfere with each other.
- The security policy between different security policy domains is suitably mediated.
- The security mechanisms themselves cannot be tampered with.
- The security policy data cannot be changed except by authorized administrators.
- The system cannot be put into an undefined or insecure state as a result of the operation of nonprivileged code.

The distributed TCB also needs to provide the required information so that applications can enforce their own security policies in a way that is consistent with the domain security policy.

.

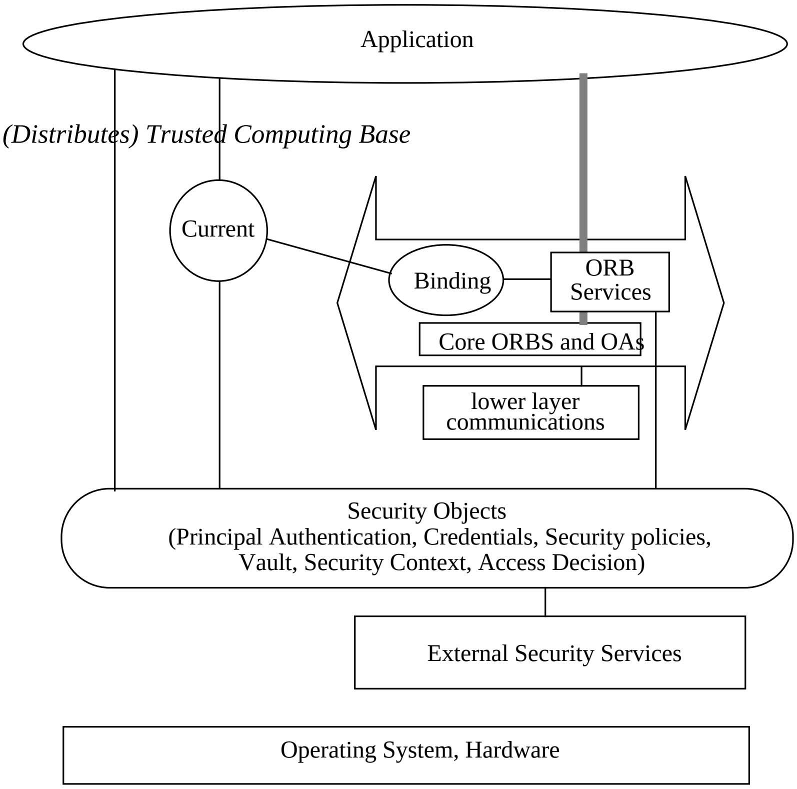

Application
(Distributes) Trusted Computing Base
Current
Binding
ORB Services
Core ORBS and OAs
lower layer communications
Security Objects
(Principal Authentication, Credentials, Security policies,
Vault, Security Context, Access Decision)
External Security Services
Operating System, Hardware

Figure E-2 Distributed TCB

The TCB in an OMA-compliant secure system is normally distributed and includes components as follows.

- The distributed core ORBs and associated Object Adapters
  Core ORBs are trusted to function correctly and call the ORB Security Services correctly in the right order, but do not need to understand what these do.
  Object Adapters are trusted to utilize the operating system facilities to provide the required protection boundaries between components in line with the security policy.
- The associated ORB Services
  ORB Services other than security are trusted similarly to the ORB. ORB Security Services are used to provide the required security on object invocation.
- Related objects
  ORB Services use objects such as the binding and Current to find which security is required.
- Security objects
  Security objects include those available to applications such as Principal Authentication and Credentials and those called by security interceptors (Vault,

Security Context, Access Decision, and Security Audit). These are trusted to function correctly to enforce security in line with the security policy and other requirements.

- Any external security services used by the security services, as part of enforcing the security policy.
- The supporting operating systems.
  These are trusted to ensure that objects (in different trust domains) cannot interfere with each other (using protection domains). The security services should also ensure that the security information driving the security policy (such as the credentials and security contexts) is adequately protected from the application objects using such features.
- Optionally, lower layer communications software. However, this does not generally need to be particularly secure (at least for normal commercial security) as protection of data in transit is done by the security association and message protection interceptors, which are independent of the underlying communication software.

A distributed system may be split into domains, which have different security policies. These domains may include ORBs and ORB Services with different levels of trust. Trust between domains needs to be established, and an interdomain policy between them enforced. The ORB security services (and external security services that these call) to provide this interdomain working are part of the distributed TCB. Note, therefore, that the parts of this TCB in different domains may have different levels of trust.

Note that application objects may enforce their own security polices, if these are consistent with the policy of the security domain. However, failure to enforce these securely will affect only the applications concerned and any other application objects that trusted them to perform this function.

E.3.2.1 Protection Boundaries

The general approach is to establish **protection boundaries** around groups of one or more components, which are said to belong to a corresponding **protection domain**. Components belonging to a protection domain are assumed to trust each other, and interactions between them need not be protected from each other, whereas interactions across boundaries may be subject to controls. Protection Boundaries and Domains are a lower level concept than Environment Domains; they are the fundamental protection mechanism on which higher levels are built.

At a minimum, it must be possible to create protection boundaries between:

- Application components that do not trust each other.
- Components that support security services and other components.
- Components that support security services and each other.

E.3.2.2 Controlled Communications

As well as providing protection boundaries, it is necessary to provide a controlled means of allowing particular components to interact across protection boundaries (for example, an application invoking a Security Object (explicitly), or an interceptor (implicitly).

It must not be possible for applications to bypass security services which enforce security policies. It is therefore necessary to ensure that the components supporting those services are always invoked when required. This is achieved by using both protection boundaries and controlled communications to ensure that client requests (and server responses) are routed via the components (interceptors and Security Objects), which implement the security services.

Figure E-3 illustrates the segregation of components implementing security services into separate protection domains from application components; the only means of communication between components is via controlled communication paths.

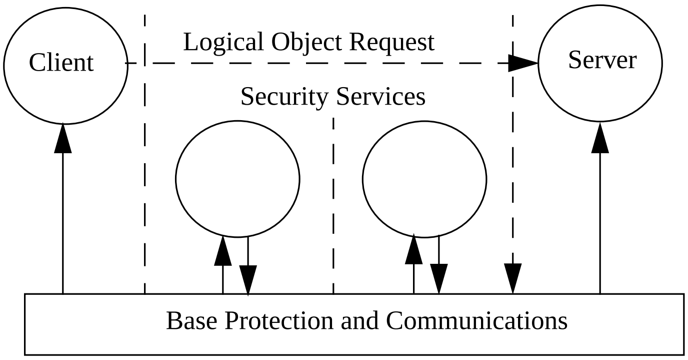


Figure E-3 Base Protection and Communications

In implementation terms, components could, for example, be executed in separate processes, with process boundaries acting as protection boundaries. Alternatively, security services could be executed in-process with (i.e., in the same address space as) corresponding client and server application components, provided that they are adequately protected from each other -- for example, by hardware-supported multilevel access control mechanisms).

Figure E-4 shows two examples of protection boundaries. In the first example, the boundaries between components might be process boundaries. In the second example, ORB and security components might be protected from applications by memory protection mechanisms (e.g., kernel and user spaces) and client and server components might be protected from each other by physical separation.

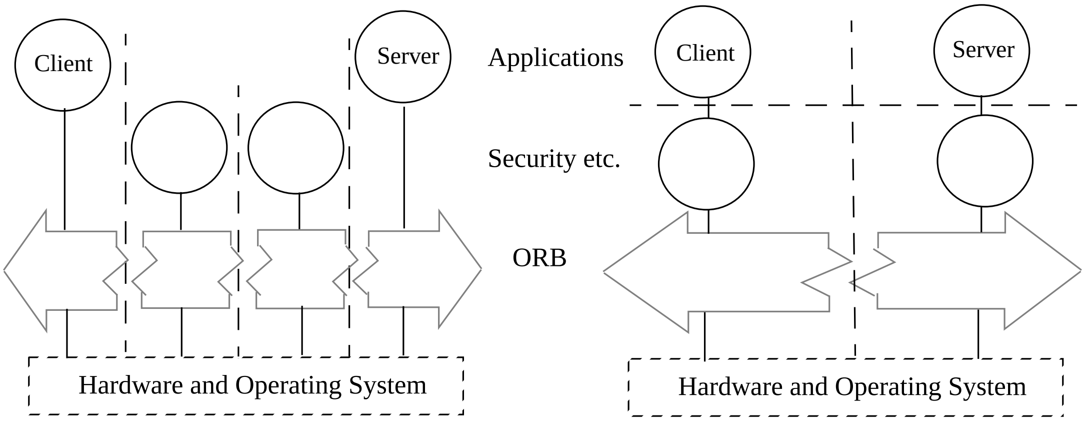


Figure E-4 Protection Boundaries

E.3.3 Construction Options

For some systems, the TCB in domains of the distributed system may need to meet security evaluation criteria for both functionality and assurance (in the correctness and effectiveness of the security functionality) as defined in TCSEC, ITSEC, or other security evaluation criteria.

The split into components previously described allows a choice over the way the system is constructed to meet different requirements for assurance and performance.

This section describes three options for how the system may be constructed, as follows:

- A commercial system where all applications are generated using trusted tools.
- A commercial system with limited security requirements.
- A higher security system.

Note – These are just examples to show the type of flexibility provided by the security model. It is not expected that any implementation will provide all the options implied by these.

Example Using Trusted Generation Tools and ORBs

If all applications are generated using trusted tools, applications can be trusted not to interfere with other components in the same environment. Therefore there is no need to provide protection boundaries between different application objects or between application objects and the underlying ORB.

If the ORB and ORB Services are also trusted, there may need be no need to provide a protection boundary between the ORB and the underlying security services and objects. It may well be acceptable to run them all in the same process, relying on the trust between the components, rather than more rigidly enforced boundaries.

However, if the application generation tools and the ORB are less trusted than the security services, then there may need to be a protection boundary to prevent access to security-sensitive information in the Credentials, Security Context, and Vault objects.

Commercial System with Limited Security Requirements

Some systems may not contain very sensitive business information, so enterprises may not be prepared to pay for a high level of security. They may also know that the probability of serious malicious attempts to break the system is low, and decide that protecting against such attempts is not worth the cost. They may also choose not to sacrifice performance for better levels of security.

In many systems, applications are generated using tools that are not particularly trusted. For example, using a C compiler, it would be possible to write an application that can read, or even alter, any information within the same protection domain. Theoretically, providing good security implies putting protection boundaries between each application object, and between applications and the ORB and Security Services.

The security model allows environment domains to be defined, where enforcement of policy can be achieved by means local to the environment. For example, objects in the same identity domain can share a security identity. Applications belonging to environment domains may trust each other not to interfere with each other, and so can be put in the same protection domain.

It may also be acceptable to run (part of) the ORB in the same protection domain as the application objects. This assumes that an interface boundary between applications and the ORB is sufficient protection from accidental damage (the probability of an application corrupting an ORB being low in a commercial system). Even if the application does corrupt the ORB, damage is limited, as the ORB does not handle security-sensitive data.

In some commercial systems, it may also be acceptable to run some of the security services in the same protection domain as the application and ORB. The chance of these being accidentally (or maliciously) corrupted may be low, so it may be acceptable to risk a failure to enforce the access control policy because the Access Decision object is corrupt.

However, it will often be desirable to protect the state information of security objects, which contain very sensitive security information from the applications.

Higher Security System

In a security system requiring high assurance, different security policies may be used. For example, label-based access controls may be used and these may be mandatory (set under administrator’s controls) and not changeable by application objects.

Stronger protection boundaries are also likely to be needed, allowing:

- Individual applications to be protected from each other. Even if environment domains are used, the size of the domain is likely to be smaller.
- The ORB and ORB Services to be protected from the application.
- The core security objects, which contain security-sensitive information such as keys to be protected from applications and ORBs, etc.
- Particular secure objects (e.g., the Access Decision objects) to be separate from others, as they may have been written by someone less trusted than those who wrote, for example, the Security Context objects.

E.3.4 Integrity of Identities (Trojan Horse Protection)

In traditional procedural systems, protecting the integrity of an identity is straightforward; programs are stored in files, which are protected against modification by operating system access control mechanisms. When invoked, programs run inside a process whose address space is protected by operating system memory protection mechanisms. Programs load code in fairly predictable ways.

Since this specification does not mandate which entities have identities, implementors have a wide variety of choices; identities may be associated, for example, with the following:

- Object instances
- Servers
- Object adaptors
- Address spaces

If identities are associated with object instances, precautions are necessary to prevent object instance code from being modified by other code (which may have no identity, or a different identity) in the instance's address space.

Servers may permit dynamic instantiations of previously unknown classes into their address spaces. This makes it difficult to determine what code is running under an identity if identities are associated with servers; this in turn makes it difficult to determine whether a server identity can be "trusted." Identified servers must therefore be provided with some way of controlling what code can run under their identities.

Observing the following guidelines will help to ensure integrity of identities.

- Code running under one identity must not be permitted to modify code running under another identity without passing an authorization check.
- It must be possible for an identified "entity" to control which code runs within the scope of its identity.

E.4 Guidelines for Application Interface Model

This section provides architecture and implementation guidelines for the application interface model of the CORBA security architecture described in Section 2.2, “Security Architecture,” on page 2-28. The security functions provided in the model and the basis for trust are described.

E.4.1 Security Functions

E.4.1.1 Logging onto the System

When a user or other principal wants to use a secure object system, it authenticates itself and obtains credentials. These contain its certified identity and (optionally) privilege attributes, and also controls where and when they can be used. This principal information is integrity-protected and it should be possible to ascertain what security service certified them.

Walkthrough of Secure Object Invocation

The following is a walkthrough of what happens when a client invokes a target object.

- The client invokes the object using its object reference. The ORB Security Services are transparent to the client and application object and use the security information with the object reference and the security policy to decide on the security facilities required. There are separate ORB Services for security associations, message protection, and access control on object invocation, but the audit service can be called by any or none of these according to security policy.

  The client and target object establish the required level of trust in each other, transmitting security tokens to each other to provide the required degree of proof. For example, they may or may not require mutual authentication. It is expected that most security mechanisms will provide options here, though the details of how they do this, and the form of tokens used, is mechanism dependent.

  The principal’s credentials are normally passed from client to target object transparently. These should be protected in transit from theft and replay as well as for integrity of the information itself (though some security mechanisms may not support this). The Vault object will validate these, checking that it trusts who certified them, as well as whether they are still intact.

  Different ORB services may be called at the target end. For example, access control is normally called at the server, rather than the client.

- Once the security association has been established between client and target object, the request can be passed using the message protection interceptor to protect it. This should be able to provide integrity and/or confidentiality protection. It should also be able to provide continuous authentication, as the messages will be protected using keys only known to this client and server (or the trust group for the target object).

- The application object may also call security services for access control and audit. These will use the security information available from the environment to identify the initiating principal and its privileges.
- This application object may now act as a client, and call further objects. It may delegate the client's credentials or use its own (or use both). However, there may be constraints on whether the client's credentials can be delegated. For example, a particular principal's credentials may be constrained to particular groups of objects.

E.4.2 Basis of Trust

Users have some trust in application objects, and application objects have some trust in other objects. Both may:

- Trust application objects to perform the business functions.
- Have limited trust in some applications, or domains of the distributed system, so restrict which of their privilege attributes are available to these objects.
- Want to restrict the extent that their credentials can be propagated at all.
- Have to prove their identity to the system so it can enforce access on their behalf, unless they are only going to access publicly available services.

Both users and applications trust the underlying system to enforce the system security policy, and therefore protect their information from unauthorized access and corruption.

E.5 Guidelines for Administration Model

This section provides architecture and implementation guidelines for the administration model of the CORBA security architecture described in Section 2.2, "Security Architecture," on page 2-28. The security functions provided in the model and the basis for trust are described.

E.5.1 Security Functions

Object and Object Reference Creation

When an object is created in a secure object system, the security attributes associated with it depend on the security policies associated with its domain and object type, though the object may be permitted to change some of these. These attributes control what security is enforced on object invocation (or example, whether access control is needed and, if so, the Access Decision object to be used; the minimum quality of protection required).

The object reference for a such an object is extended to include some security information. For example, it may contain:

- An extended identity. This includes the object identity as normal in an object reference. However, it will also contain the identity of the trust domain, if the object belongs to one. Small objects, which are dynamically created and do not need to be protected from each other, will normally share a trust domain. There could also be a node identity.
- Security policy attributes required by the object when invoked by a client such as the minimum quality of protection of data in transit.
- The security technology it supports. It may also contain some mechanism-specific information such as its public key, if public key technology is being used, and particular algorithms used.

Much of the information is just “hints” about which security is required, and will be verified by the ORB services supporting the target object, so does not need protecting.

E.5.2 Basis of Trust

Authorization Policy Information

Domain objects may store policy information inside their own encapsulation boundaries, or they may store it elsewhere (for example, authorization policy information could be encapsulated in the state data of the protected objects themselves, or it could be stored in a procedural Access Control Manager whose interfaces are accessible to Domain objects). Wherever authorization policy information is stored, it must be protected against modification by unauthorized users.

Authorization policy information must be modifiable only by authorized administrators.

Audit Policy Information and Audit Logs

Audit policy information is security sensitive and must be protected against unauthorized modification. Audit logs are security sensitive and may contain private information; they should be viewed and changed only by authorized auditors.

- Audit policy information must be modifiable only by authorized audit administrators.
- Audit logs must be protected against unauthorized examination and modification.

E.6 Security Object Implementation Model

E.6.1 Guidelines

This section provides architecture and implementation guidelines for the security object implementation model of the CORBA security architecture described in Section 2.2, “Security Architecture,” on page 2-28. The security functions provided in the model and the basis for trust are described.

E.6.2 Security Functions

The distributed core ORBs, object adapters, ORB security services, and security objects provide the underlying implementation to support the application and administration interfaces.

E.6.3 Basis of Trust

Target Object Identities

CORBA objects do not have unique identities; for this reason, when objects that are not associated with a human user authenticate themselves in a secure CORBA system, they use “security names.” Successful authentication to a target object indicates that it possesses the authentication data (perhaps a cryptographic key), which is presumed to be known only to the legitimate owner of the security name. An object’s security name may be included in references to that object as a “hint.” The question “how do applications know that the security-name hint is reliable?” naturally arises.

The answer is as follows:

- If the **EstablishTrustinTarget** security feature is specified, then the security services defined in this specification will authenticate the target security name found in the target object reference. The semantics of this authentication operation include an assumption that the security name in the reference corresponds to an identity that the user is willing to trust to provide the target object’s implementation. There is no way for the security services to test this assumption.
- If your implementation provides a trusted source of object references, then everything will work properly. If you do not have a source of trusted object references, the specification provides a **get_security_names** operation on the object reference through which applications can retrieve the target’s security name and perform any tests, which may help satisfy them of its validity.

CORBA object references can circulate very widely; for example, they can be “stringified” and then (potentially) copied onto a piece of paper. Implementations with very high integrity requirements could ensure that references are trustworthy by providing a trustworthy service that generates references and cryptographically signs the contents, including the target security name.

Assumptions about Security Association Mechanisms

Implementation of a secure CORBA system requires use of security mechanisms to enforce the security with the required degree of protection against the threats. For example, cryptographic keys are normally used in implementing security, for functions such as authenticating users and protecting data in transit between objects. However, different security mechanisms may use different types of cryptographic technology (e.g. secret or public key) and may use it in different ways when, for example, protecting data in transit. These cryptographic keys have to be managed, and again, the way this is done is mechanism specific.

A full analysis of how well an implementation counters the threats requires knowledge of the security mechanisms used. However, this specification does not dictate that a particular mechanism is used.

It does assume that the security mechanisms used for authentication and security associations can provide the relevant security countermeasures listed in Section E.2.4, “Countermeasures,” on page E-7. These are expected to be provided by a number of security mechanisms, which will be available for protecting secure object systems. Therefore, the analysis of threats and the trust model assume this facility level.

It would be possible to use a security mechanism that does not provide some of these facilities (for example, mutual authentication, or even to switch this off to improve performance in systems that can provide it). However, if such a system is used, it will be vulnerable to more threats.

Invoking Special Objects

Some of the objects described in this document are **locality constrained** objects, which bypass the normal invocation process and therefore are not subject to the security enforced by the ORB services. The **Current** object (used, for example, by the target object to obtain security information about the client) is of this type. Protection of these objects is provided by other means, for example, using protection boundaries previously described.

E.6.4 Basis For ORB Assurance

The ORB must function correctly (e.g., when enforcing security policy on object invocation and object creation as defined in this specification). Likewise the underlying host platforms must function correctly in their provision of the security mechanisms employed, and relied upon, by the ORB. Both must do this to the level of assurance specified in its Conformance Statement (which is described in Appendix E). This section identifies many of the most critical design considerations related to providing these assurances in a DOC system.

Isolating Security Mechanisms

Figure E-5 depicts how security functionality and trust is distributed throughout the architecture.

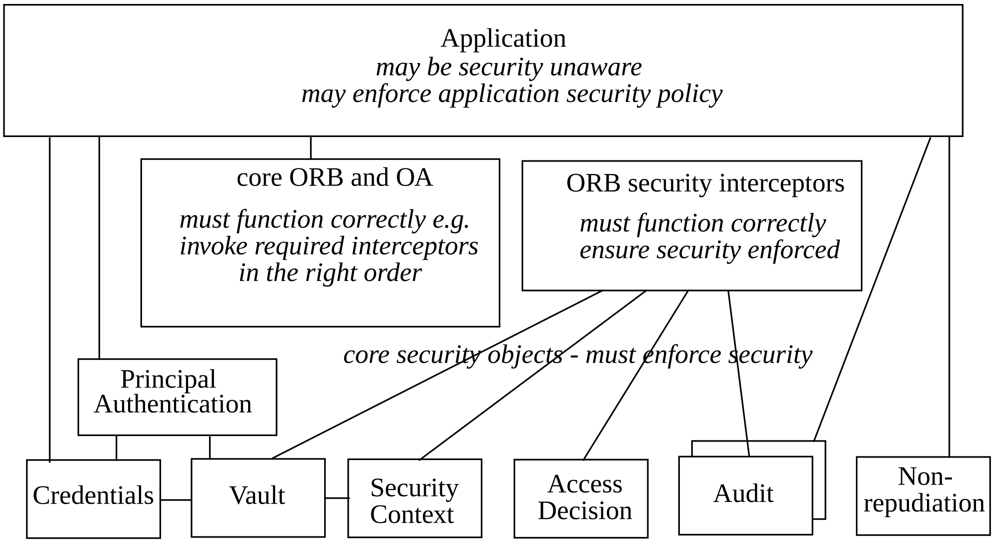

Application
may be security unaware
may enforce application security policy
core ORB and OA
must function correctly e.g. invoke required interceptors in the right order
ORB security interceptors
must function correctly ensure security enforced
core security objects - must enforce security
Principal Authentication
Credentials
Vault
Security Context
Access Decision
Audit
Non-repudiation

Figure E-5 Distribution of Security Functionality and Trust

The split of security objects is designed to reduce (as much as possible) the amount of security-sensitive information, which must be visible to applications and ORBs.

- Only log-in applications (where provided) need to handle secrets such as passwords, and then only briefly during authentication.
- Cryptographic keys and other security-sensitive information about principals are held with Credentials objects. References to Credentials objects are visible to applications so they can invoke operations on them to, for example, reduce privileges in the credentials before calling an object. However, no operations on the Credentials provide visibility of security information such as keys.
- Security information used to protect application data in transit between objects is held in Security Context objects, which are not visible to applications at all. (Target applications can ask for attributes associated with an incoming invocation using the Current object.)

Security objects such as Credentials, Security Context, and Access Decision objects are also not used directly by the core ORB, only by the security interceptors. Therefore the core ORB needs to be trusted to call the interceptors correctly in the right order, but does not need to understand security or have access to the security-sensitive information in them.

The split also is intended to isolate components which may be replaced to change security policy or security mechanisms. For example, to replace the access control policy, the Access Decision objects need to be changed. However, the access control interceptor will remain responsible for finding and invoking the right Access Decision object. To replace the security mechanisms for security association, only the Vault and associated Security Context objects need to be replaced.

Integrity of the ORB and Security Service Objects

Security in a CORBA environment depends on the correct operation of the ORB and Security Services. In order for these mechanisms to operate correctly, the following rules must be followed:

- The ORB and Vault code must not be modifiable by unauthorized users or processes.
- The ORB must protect all messages, according to policy, using the message protection interfaces.
- The ORB must always check the client's authorization before dispatching a client's message to a protected object.

Safeguarding the Object Environment

To guard against unauthorized modification of the ORB and security services, implementors should use Operating System protection mechanisms to isolate the ORB and Security Service objects from untrusted applications and user code.

Note that some modifications of ORB or Vault code may not compromise system integrity. For example, in a CORBA implementation, which relies on third-party authentication and does not share Vault or ORB objects between processes, corruption of the client-side Vault (or ORB) by user-written code may not compromise system security. (This is because the client-side ORB and Vault in a third-party-based system may, depending upon the implementation, contain only information that the user is entitled to know and change anyway. In this case, nothing the user can do to information on his machine will enable him to deceive the third-party authentication server about his identity and credentials.)

Safeguarding the Dispatching Mechanism

To ensure that the ORB always checks the client's authorization before dispatching a client's message to a protected object, ORB implementors should follow one of the following rules:

- Eliminate "direct dispatching" mechanisms (which permit clients to dispatch messages directly to target objects without going through the ORB).
- Permit "direct dispatching" only after checking authorization and issuing "restricted object references" to client objects. A "restricted object reference" is one that grants access only to those methods of the target object, which the client is authorized to invoke.

Safeguarding Information in Shared Vault Objects

Vault objects encapsulate identity-specific, security-sensitive information (for example, cryptographic keys associated with Security Context objects). If code owned by one principal can penetrate a Vault object and examine or modify another principal's information, security can be compromised.

In an implementation that does not permit sharing of Vault objects by multiple identities, this problem does not arise. However, if Vault objects are accessible to and encapsulate information about multiple identities, the following guidelines should be observed:

- Do not permit a Vault object, which encapsulates one principal’s Security Contexts, to exist in the same address space as code running under a different principal’s identity.
- If a Vault object contains Security Contexts for two different principals, ensure that no principal is able to obtain or use another principal’s Security Contexts.

Facilities Not In This Specification F

F.1 Introduction

Security in CORBA systems is a big subject, which affects many parts of the Object Management Architecture. It was therefore decided to phase the specification in line with the priorities agreed as part of the security evaluation criteria by the Security Working Group prior to the production of this specification.

This specification therefore includes the core security facilities and the security architecture to allow further facilities to be added. Priority has been given to those requirements most needed by commercial systems. Even with these limitations, the size of the specification is larger than desirable for OMG members to review easily or for vendors to implement.

Some of the facilities omitted from this specification are agreed to be required in some secure CORBA systems, and so are expected to be added later, using the usual OMG process of RFPs to request their specification.

This appendix lists those security facilities which are not included in the specification, but left to later specifications, which may be in response to further RFPs for Object Services or Common Facilities.

F.2 Interoperability Limitations between Unlike Domains

Secure interoperability is included in this specification. This allows applications running under different ORBs in different domains to interoperate providing that:

- Both support and can use the same security mechanisms (and algorithms, etc.) for authentication and secure associations (an ORB may support a choice of security mechanisms).
- Use of these between the domains will not contravene any government regulations on the use of cryptography.

- The security policies they support are consistent -- for example, use the same types for privileges which can be understood in both places.

Limitations in the specification which affect this type of interoperability are:

- The standard policies defined do not include specifying different policies when a client communicates with different domains (though it is possible to define specific policies to do this).
- There is no specification of the mapping policies required to translate attributes when crossing a domain boundary where these policies are inconsistent, and how these must be positioned, for example, to allow delegation of the mapped attributes. Again, such mapping policies are not prevented.
- In general, there is no specification of how federated policies are implemented.
- There is no specification of gateways to handle interoperability between security mechanisms. It is expected that only limited interoperability between particular security mechanisms will ever be provided, so this is not expected to be the subject of an RFP in the foreseeable future.

F.3 Non-Session-Oriented SECIOP Protocol

The SECIOP protocol defined in Section 3.2, “Secure Inter-ORB Protocol (SECIOP),” on page 3-34, assumes that all underlying security mechanisms are session-oriented. The current specification does not support security mechanisms, which encapsulate key distribution and other security context management information in a single message along with the data being protected (examples of such mechanisms include those accessed through the proposed internet IDUP-GSS-API interface). Changes to the SECIOP protocol would be required to support non-session-oriented protocols.

F.4 Mandatory Security Mechanisms

The current specification does not mandate any particular security mechanism which all secure ORBs must implement. This is because the submitters did not think it was possible to specify out-of-the-box interoperability adequately in the timescale of this submission.

F.5 Specific Security Policies

This specification includes some standard types of security policies for security functionality such as access control, audit, and security of invocations. These are aimed at general commercial users. Some enterprises may require other types of policies, for example, support of mandatory access controls. Where there is a sufficient market for such policies, new policies may be defined, providing they fit with the replaceability interfaces defined in this specification.

F.6 Other Audit Services

This specification only contains limited audit facilities, which allow audit records of security relevant events to be collected. It does not include:

- Filtering of records after generation to further reduce the size of the audit trail.
- Routing audit records to a collection point for consolidation and analysis or routing some as alarms to security administrators. (However, routing may be done using the OMG Event Service, if that is secure enough.)
- Audit reporting or analysis tools to use the audit trails to track down problems.

F.7 Possible Enhancements

F.7.1 SECIOP Mechanism and Option Negotiation

This specification assumes the mechanism identifiers in the IOR allow the client to choose what mechanisms and options to use when communicating with this target. Therefore, it does not define protocol exchanges to allow the client and target to negotiate either mechanisms or options.

However, if the target supports a number of mechanisms and options, the size of the IOR could become larger than desirable. So in future, it may be desirable to define protocol exchanges for mechanism negotiation, for example, using [19].

F.7.2 Further Key Distribution Options

The current CSI-ECMA protocol defines secret and public key options for key distribution and a hybrid option where secret keys are used within a domain, but public keys are used between domains. It does not define the protocol for use in the sort of hybrid system where the initiator uses secret key and target uses public key technology and vice versa.

This may be needed for interoperation between unlike domains. If so, further architectural options from ECMA 235 may need to be included in the specification.

F.7.3 Further Delegation Options at/above Level 2

The current level 2 specification supports restricting where an initiator's attributes can be used to targets identified by security name. Further options for restricting where a PAC may be delegated could be added (e.g., to restrict delegation to particular delegation policy domain). This would require definition of further "qualifier attributes" in the CSI-ECMA protocol (see application trust groups in ECMA 235). It would also require administration of this, which would best be done by extending the security policy administration in Section 2.4, "Administrator's Interfaces," on page 2-116.

Composite delegation of the initiator plus immediate invoker kind is described in the CSI protocol, but is not mandatory at level 2. Further composite delegation options, including traced delegation, could be added.

F.8 Interoperability when using Non-Repudiation

The optional Non-repudiation service in the CORBA Security specification generates NR tokens. This specification does not specify the technology used to generate these tokens or a standard form for them. Interoperability of evidence tokens would require a standard specification for such tokens.

This CSI specification is focused at inter-ORB interoperability, and therefore the IOR and SECIOP protocol. So it also does not specify the format of evidence tokens as they do not affect the SECIOP protocol. However, these evidence tokens may be passed between ORBs as parameters, and will not be understood by an ORB which does not use the same security technology.

In future, a mandatory interoperability evidence token format should be defined, at least for a limited number of types of evidence. This is expected to be compatible with the public key mechanism specified in this document and use X.509 version 3 certificates.

F.9 Audit Trail Interoperability

The CORBA Security specification includes an Audit Channel interface which allows applications and ORBs to write records to the audit trail. The way this Audit Service routes the audit records is not defined. This could be done using the OMG Event Service or other means. Also, the stored/on-the-wire format of audit records is not defined.

So there is no standard OMG defined method of bringing together audit records from different Audit Services.

F.10 Management

This specification contains only the management interfaces which are essential for security policy management. It specifies how to obtain and use security policy objects. However, it does not contain:

- Specification of facilities for handling domains, policies other than those required for security policy administration.
- Specification of facilities for the management of some aspects of security. For example, it does not specify how to create and install permanent keys, as this is implementation specific.

F.11 Reference Restriction

This specification requires the movement of credentials to delegate access rights from one object to another. Another technique of access rights delegation restricts the use of an object reference according to a set of criteria. This approach, know as reference restriction, is under study by a number of vendors, but is not ready for standardization at this time. The criteria used to restrict references could include:

- Whether an object has the right to assert certain privileges, such as act on behalf of a principal, act on behalf of a group of principals, act in a particular role, act with a particular clearance, etc.
- Whether the object reference has been limited to use within a given time interval.
- Whether a particular method can be used by an object holding the object reference.

Various techniques for restricting object references have been developed. Some use cryptographic methods, while others store state in the object associated with the restricted reference, allowing the object to decide if a method request meets the restricted reference use criteria.

It is anticipated that vendors will explore this type of access rights delegation and move towards the standardization of an interface supporting it in a submission to a future RFP.

F.12 Target Control of Message Protection

In the current specification, message protection can be specified by policy administration at both the client and the target object.

Requesting an operation on an object may result in many other objects being invoked. The CORBA security specification in this document allows an intermediate object in such a chain of objects to delegate received credentials to the next object in the chain (subject to policy). However, the current specification does not allow the application to control when and where these credentials are used. A later specification may provide such controls to ride the default quality of protection selectively. Therefore, it could cause some messages to have different qualities of protection during a security association.

The target has no equivalent interface to request the quality of protection for a particular response. There are cases where this could be useful.

A future security specification should consider adding control of quality of protection by the target for individual responses.

F.13 Advanced Delegation Features

Requesting an operation on an object may result in many other objects being invoked. The CORBA security specification in this document allows an intermediate object in such a chain of objects to delegate received credentials to the next object in the chain (subject to policy).

However, the current specification does not allow the application to control when and where these credentials are used.

A later specification may provide such controls.

If so, it is expected that a **set_controls** operation on the Credentials object will be added to enable the application to set the controls, and a matching **get_controls** operation to enable it to see what controls apply (see the **set_privileges** and **get_attributes** operations defined in Interfaces under Section 2.3.4, “The Credentials Object,” on page 2-78).

The **set_controls** operation would allow the application to specify a set of required control values such as delegation mode (allowing for richer forms of delegation), restrictions on where the credentials may be used and/or delegated, and validity period.

Note: These operations were not included in the specification because of concerns about portability of applications using them. Current delegation implementations use a wide variety of delegation controls, and some use similar controls in semantically different ways. Further implementation experience and investigation may make it possible to define a portable, standard set.

F.14 Overlapping and Hierarchical Domains

This specification does not require support for overlapping or hierarchical security policy domains. However, it is possible to implement both using the interfaces provided.

Recall from Section 2.4, “Administrator’s Interfaces,” on page 2-116, that the DomainAccessPolicy for each domain defines which rights are *granted* to subjects when they attempt to access objects in the domain. In order to make an access decision, the AccessDecision logic also needs to know which rights are *required* to execute the operations of an object, which is a member of the relevant domain. The RequiredRights interface provides this information; the AccessDecision object will probably use this interface in most implementations.

A **RequiredRights** instance can be queried to determine which rights a user must be granted in order to be allowed to invoke an object's operations. The intended use of **DomainAccessPolicy** and **RequiredRights** objects by the **AccessDecision** object is illustrated next, in Figure F-1.

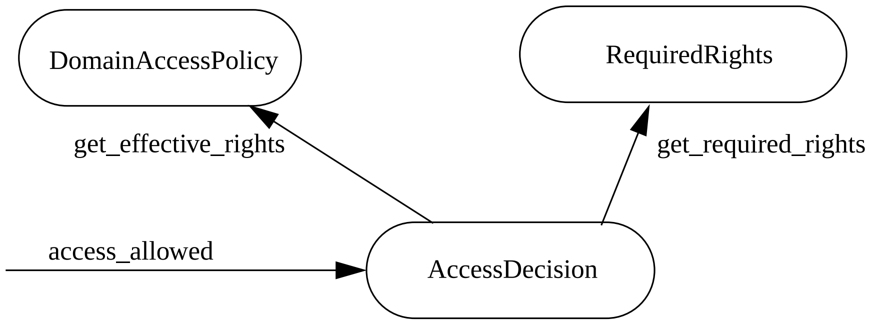


Figure F-1 Intended Use by AccessDecision

AccessDecision retrieves the relevant policy object by calling **get_domain_managers** on the target object reference, and then calling **get_domain_policy(access)** on the returned domain manager (assuming for purposes of this example that there is only one). It then calls **get_effective_rights** on the returned policy object. **AccessDecision** then calls **get_required_rights** on **RequiredRights** and compares the returned list of required rights with the effective rights. If all required rights have been granted, it grants the access.

Figure F-2 on page F-8 illustrates how the specification could be implemented to support overlapping access policy domains (i.e., to allow an object to be a member of more than one domain, such that each domain has an access policy and all domains' access policies are applied). In the diagram, the **AccessDecision** object must have logic to combine the policies asserted by the various **AccessPolicy** objects (which may involve evaluating which **AccessPolicy** object's policy takes precedence over the others). Note that the **AccessDecision** object knows the target object reference, because it is passed as an input parameter to the **access_allowed** operation.

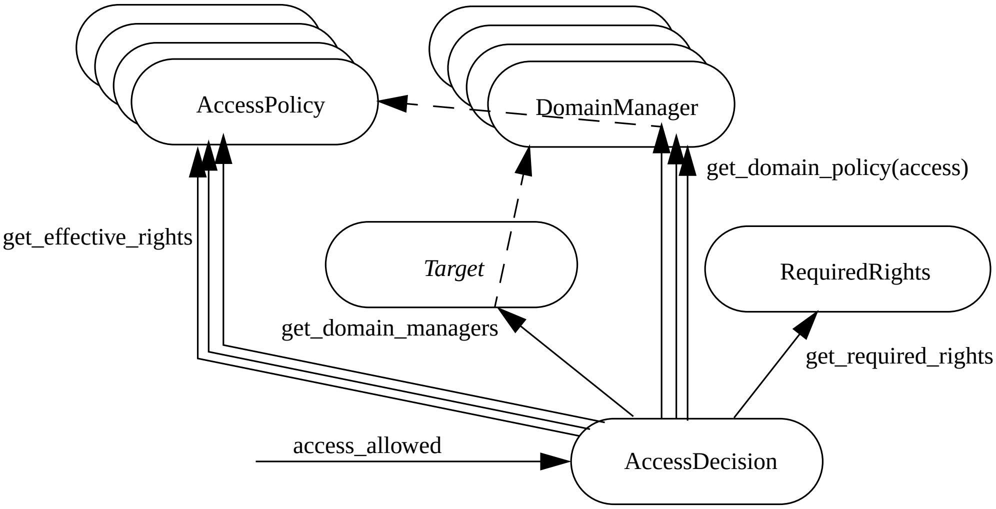


Figure F-2 Supporting Overlapping Access Policy Domains

Hierarchical domains can be handled in a similar way as illustrated in Figure F-3 (note that once again the **AccessDecision** object's implementation is responsible for reconciling the various retrieved policies).

.

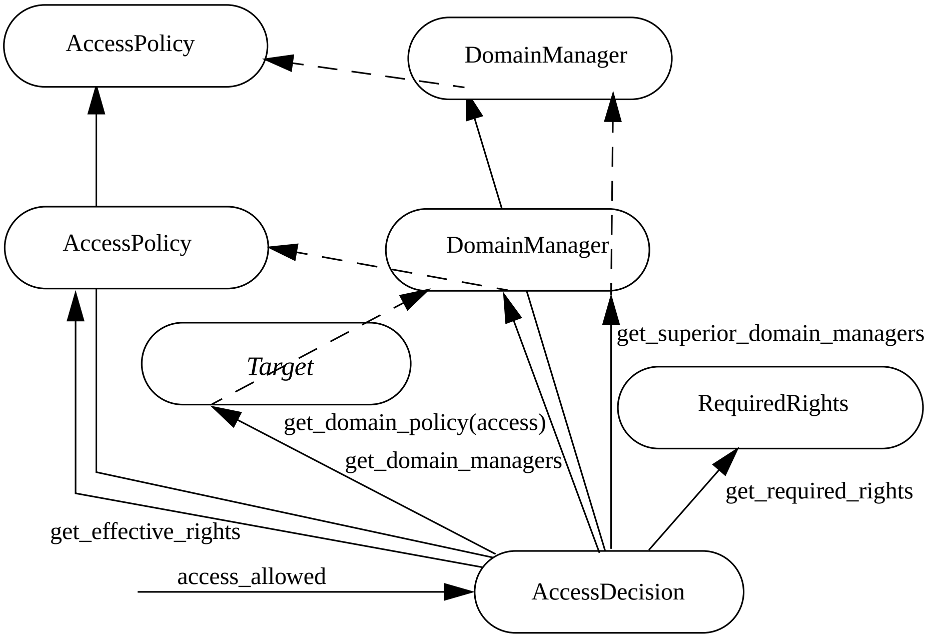


Figure F-3 Hierarchical domains

F.15 Capability-Based Access Control

Capability-based systems store access policy information in tokens, which are passed from sender to receiver along with a message, rather than in tables associated with target objects or domains. In such systems, the **DomainAccessPolicy** object will generally not be used in resolving target-side access control checks. Instead, a **CapabilityAccessPolicy** object might be returned from a call to **Object::get_policy** in a capability-based system. This object could retrieve the granted rights from the capability (which will be associated with the requester's credentials), illustrated in Figure F-4.

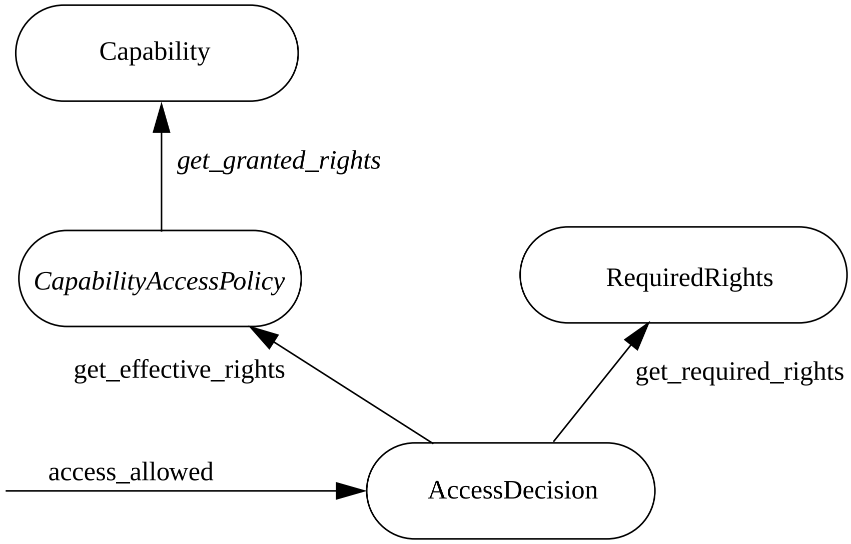


Figure F-4 Retrieving Granted Rights

Note that neither the CapabilityAccessPolicy interfaces nor the Capability interfaces are defined in this specification (the **get_granted_rights** call to the capability in the previous diagram is printed in italics, to indicate that no IDL is provided for it in this specification). The diagram assumes that **CapabilityAccessPolicy** inherits the **get_effective_rights** operation from **AccessPolicy**.

F.16 Non-repudiation Services

This specification contains Non-repudiation Services for evidence handling. It is anticipated that future service offerings could include data protection processing and the specification of a delivery service. In addition, it is expected that policy processing interfaces will emerge to cover the broad range of non-repudiation policy coverage within the service.

It is anticipated that the data protection and delivery service functions will be reaching a level of maturity within other standards domains (such as IETF and ISO SC27), which should allow a richer definition of these services to be enabled in future revisions of this specification.

The absence of these services in this specification means that application writers and manipulators will need to consult local implementation practice for the correct course of action to be taken when writing or porting their software.

This specification also does not include a standard format of evidence token for interoperability. In the future, a token format based on public key certificates may be specified.

Interoperability Guidelines G

G.1 Introduction

This appendix includes:

- Guidelines for defining Security Mechanism TAGs in Interoperable Object References (IORs).
- Examples of the secure inter-ORB protocol, SECIOP.

G.2 Guidelines for Mechanism TAG Definition in IORs

Section 3.1, “Security Interoperability Protocols,” on page 3-1, defined a prototype TAG definition for security association mechanisms. This appendix provides guidelines that specifiers of mechanism TAGs (called authors here) should follow.

In addition to registering TAGs with the OMG, authors must lodge a document that explains how the mechanism (and its associated options) is mapped to this standard. Its document should:

- Identify the “security mechanism tagged component” being described. It may be either:
  - A new component TAG for the mechanism with a set of options it can have (for example, a separate TAG for each combination of mechanism and algorithm),

  or

  - Use TAG_GENERIC_SEC_MECH and specify the mechanism OID (for use in the **security_mechanism_type** field) being described by this specification.

  It may not be both.
- Specify the scope implied by the above mechanism identifier. This should not exceed:
  - Security association mechanism

    - Negotiation protocols
    - Cryptographic algorithms
    - Authentication method (e.g., public key)
- For the first example under the first bullet, describe the format, contents, and encoding of the **component_data** field for the TAG-specific components. For the second example under the first bullet, describe the format, contents, and encoding of the data in the **mech_specific_data** and components fields of the TAG specific components. In each case, this may include:
    - Allocating new component TAGs and describing the format, contents, and encoding of their data.
    - Specifying the use of these new tagged components, as well as other predefined tagged components within TAG-specific components.
    - Specifying the use of these new tagged components, as well as other predefined tagged components that may or should appear at the top level of the multicomponent profile.
- Describe a model that should be followed when defining future extensions or variations using the same mechanism.
- The author must define either by reference to another document, or explicitly, the format of the context tokens used by the mechanism in the SECIOP protocol.

G.3 SECIOP Examples

G.3.1 Mutual Authentication

In this example, the client wishes to authenticate the identity of the target (in addition to the targets requirement to authenticate the client) before it is prepared to send a request to the target.

The client sends an **EstablishContext** message to the target containing the client’s context id for the association, and the token required by the target to authenticate it and define the options chosen by the client for the association. The target verifies the client’s token and generates the token required by the client to authenticate the target. The target sends this token (along with the client’s context id for the association and its own) to the client in a **CompleteEstablishContext** message. When the client receives this message, it authenticates the target using the token supplied by the target and establishes the peer id as part of the context.

Having completed the establishment of the context, the client sends the request as part of a **MessageInContext** message, which includes the target’s context identifier and the integrity token for the message. When the target receives the message, it identifies the context by its identifier, checks the integrity of the message with the token, and passes the message to GIOP. When the reply is returned, it is sealed for integrity and returned to the client in a SECIOP **MessageInContext** with the client identifier for the context and the generated integrity token.

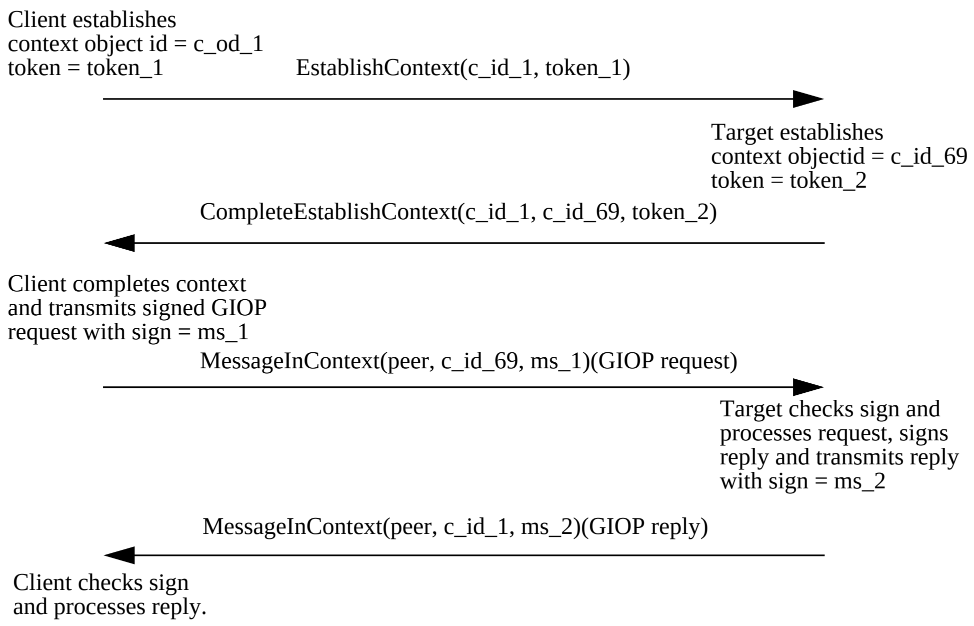

Client establishes
context object id = c_od_1
token = token_1
EstablishContext(c_id_1, token_1)
Target establishes
context objectid = c_id_69
token = token_2
CompleteEstablishContext(c_id_1, c_id_69, token_2)
Client completes context
and transmits signed GIOP
request with sign = ms_1
MessageInContext(peer, c_id_69, ms_1)(GIOP request)
Target checks sign and
processes request, signs
reply and transmits reply
with sign = ms_2
MessageInContext(peer, c_id_1, ms_2)(GIOP reply)
Client checks sign
and processes reply.

Figure G-1 Mutual Authentication

G.3.2 Confidential Message with Context Establishment

This example describes how context establishment is combined with the transmission of a confidentiality protected message when the client does not wish to authenticate the target before passing it a message.

The client establishes its context object with identifier c_id_1. This identifier is included with the token (token_1) in an EstablishContext message. The GIOP request is transformed into the message seal (ms_1) and sent with the client's context identifier in a **MessageInContext**.

When the target receives the message, it first processes the **EstablishContext** message, authenticating the client and allowing the target to create its context object. It then unseals the message in ms_1 and passes it to GIOP.

When GIOP sends the reply, SECIOP adds a **CompleteEstablishContext** message to the **MessageInContext** message, which protects the reply, to enable the target to return its context identifier to the client. When the client receives the message, it first completes its view of the context (adding the targets id to the state for the context). It can then unseal the reply from ms_2 and passes the reply message up the protocol stack.

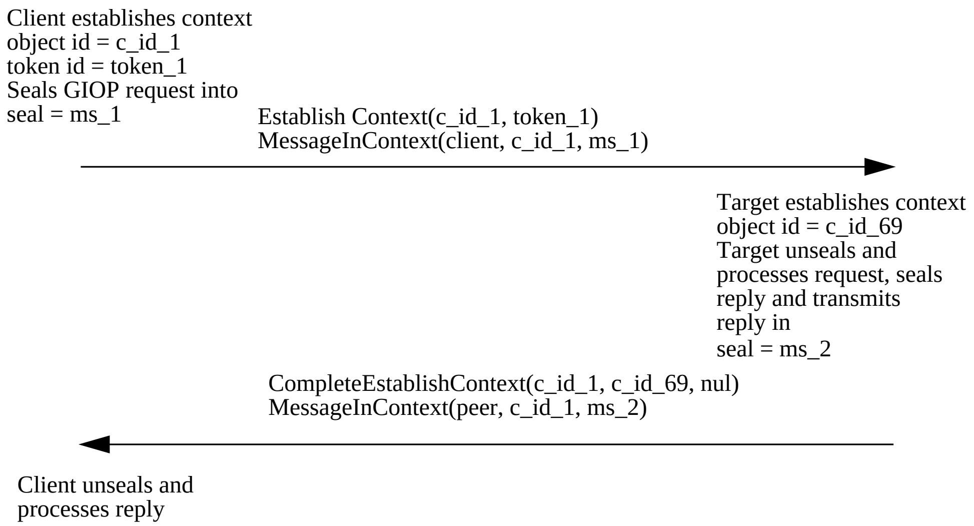

Client establishes context
object id = c_id_1
token id = token_1
Seals GIOP request into
seal = ms_1
Establish Context(c_id_1, token_1)
MessageInContext(client, c_id_1, ms_1)
Target establishes context
object id = c_id_69
Target unseals and
processes request, seals
reply and transmits
reply in
seal = ms_2
CompleteEstablishContext(c_id_1, c_id_69, nul)
MessageInContext(peer, c_id_1, ms_2)
Client unseals and
processes reply

Figure G-2 Confidential Message with Context Establishment

G.3.3 Fragmented GIOP Request with Context Establishment

In this example, the security context is established as part of the processing of a fragmented GIOP request (note that the current GIOP protocol does not support fragmentation, but this example indicates the independence of SECIOP from the current GIOP protocol and explains how the SECIOP protocol would handle a fragmented GIOP request). The sequence described reflects the requirement of the target to authenticate the client's privileges.

The client establishes its context object (with id c_id_1) and passes this identifier with the authentication token in an EstablishContext message. As the client does not require authenticating the target, this message is sent with a **MessageInContext** message with the integrity sign (ms_1) and the GIOP fragment (as the message field of the **MessageInContext**).

When the target receives the messages, it authenticates the client using token_1. It then creates a context object with c_id_69, and then processes the **MessageInContext**, checking the integrity of the message using sign ms_1. Having checked the message, it passes the fragment up the protocol stack.

The client sends the final fragment as a **MessageInContext** with sign ms_2, but as the target has not yet passed its identifier for the context to the client, the client uses its own identifier for the context.

The target finds its context object from the client's identifier (c_id_1) and checks the integrity of the message. It then passes the final fragment up the protocol stack to GIOP.

Glossary

absolute time: Time relative to the time base of 0 hours 0 minutes 0 seconds of 15 October 1582 (c.f. CORBA Time Service [3]), accurate within a known margin of error.

access control: The restriction of access to resources to prevent its unauthorized use.

access control information (ACI): Information about the initiator of a resource access request, used to make an access control enforcement decision.

access control list: A list of entities, together with their access rights, which are authorized to have access to a resource.

access decision function: The function which is evaluated in order to make an access control enforcement decision. The inputs to an access decision function include the requester's access control information (q.v.), the resource's control information, and context data.

ADO: Access Decision Object: The CORBA security object which implements access decision functions.

accountability: The property that ensures that the action of an entity may be traced uniquely to the entity.

active threat: The threat of a deliberate unauthorized change to the state of a system.

adjudicator: An authority that resolves disputes among parties in accordance with a policy. In CORBA security, an adjudicator evaluates non-repudiation evidence in order to resolve disputes.

anonymous user: A user of the system operating under a distinguished "public" identity corresponding to no specific user.

assurance: 1. Justified confidence in the security of a system. 2. Development, documentation, testing, procedural, and operational activities carried out to ensure that a system's security services do in fact provide the claimed level of protection.

asymmetric key: One half of a key pair used in an asymmetric (“public-key”) encryption system. Asymmetric encryption systems have two important properties: (i) the key used for encryption is different from the one used for decryption (ii) neither key can feasibly be derived from the other.

audit: See security audit.

audit event: The data collected about a system event for inclusion in the system audit log.

audit trail: See security audit trail.

authentication: The verification of a claimant’s entitlement to use a claimed identity and/or privilege set.

authentication information: Information used to establish a claimant’s entitlement to a claimed identity (a common example of authentication information is a password).

authorization: The granting of authority, which includes the granting of access based on access rights.

availability: The property of being of being accessible and usable upon demand by an authorized user.

call chain: The series of client to target object calls required to complete an operation. Used in this specification in conjunction with delegation.

certification authority: A party trusted to vouch for the binding between names or identities and public keys. In some systems, certification authorities generate public keys.

ciphertext: The result of applying encryption to input data; encrypted text.

cleartext: Intelligible data; text which has not been encrypted or which has been decrypted using the correct key. Also known as “plaintext”.

confidentiality: The property that information is not made available or disclosed to unauthorized individuals, entities, or processes.

conformance level: A graduated sequence of defined sets of functionality defined by the CORBA Security specification. An implementation must implement at least one of these defined sets of functionality in order to claim conformance to CORBA Security.

conformance option: A defined set of functionality which implementations may optionally provide in order to claim CORBA Security conformant functionality over and above the minimum required by the defined conformance levels.

conformance statement: A written document describing the conformance levels and conformance options to which an implementation of the OMG CORBA Security specification conforms.

control attributes: The set of characteristics which restrict when and where privileges can be invoked or delegated.

counter-measures: Action taken in response to perceived threats.

credentials: Information describing the security attributes (identity and/or privileges) of a user or other principal. Credentials are claimed through authentication or delegation (q.v.) and used by access control (q.v.).

current object: An object representing the current execution context; CORBA Security associates security state information, including the credentials of the active principal, with the current object.

DAC: Discretionary Access Control - an access control policy regime wherein the creator of a resource is permitted to manage its access control policy information.

data integrity: The property that data has not been undetectably altered or destroyed in an unauthorized manner or by unauthorized users.

DCE: Distributed Computing Environment (of OSF).

DCE CIOP:DCE Common Inter-ORB Protocol - the protocol specified in the OMG CORBA 2.0/ Interoperability specification which uses the DCE RPC for interoperability.

decipherment: Generation of cleartext from ciphertext by application of a cryptographic algorithm with the correct key.

decryption: See decipherment.

delegation: The act whereby one user or principal authorizes another to use his (or her or its) identity or privileges, perhaps with restrictions.

denial of service: The prevention of authorized access to resources or the delaying of time-critical operations.

digital signature: Data appended to, or a cryptographic transformation of. a data unit that allows a recipient of the data unit to prove the source and integrity of the data against forgery, e.g. by the recipient.

domain: A set of objects sharing a common characteristic or abiding by a common set of rules. CORBA Security defines several types of domains, including security policy domains, security environment domains, and security technology domains.

domain manager: A CORBA Security object through whose interfaces the characteristics of a security policy domain are administered.

encipherment: Generation of ciphertext from corresponding cleartext by application of a cryptographic algorithm and a key.

encryption: See encipherment.

ESIOP: Environment-Specific Inter-ORB Protocol (specified in the OMG CORBA 2.0/ Interoperability specification).

evidence: Data generated by the CORBA Security Non-Repudiation service to prove that a specific principal initiated a specific action.

evidence token: A data structure containing CORBA Security Non-Repudiation evidence.

federated domains: Separate domains whose policy authorities have agreed to a set of shared policies governing access by users from one domain to resources in another.

GSS-API: Generic Security Services- Application Programming Interface - specified by RFC 1508 issued by the Internet IETF.   An update to this interface is near completion as this is written, and it is anticipated that RFC 1508 will be superseded by a revised specification soon.

GIOP: General Inter-ORB Protocol (specified in the OMG CORBA 2.0/ Interoperability specification.)

group: A CORBA Security privilege attribute. Many users (and other principals) may be assigned the same group attribute; this allows administrators to simplify security administration by granting rights to groups rather than to individual principals.

granularity: The relative fineness or coarseness by which a mechanism may be adjusted.

hierarchical domains: A set of domains together with a precedence hierarchy defining the relationships among their policies.

identity: A security attribute with the property of uniqueness; no two principals' identities may be identical. Principals may have several different kinds of identities, each unique (for example, a principal may have both a unique audit identity and a unique access identity). Other security attributes (e.g. groups, roles, etc...) need not be unique.

immediate invoker: In a delegated call chain, the client from which an object directly receives a call.

impersonation: The act whereby one principal assumes the identity and privileges of another principal without restrictions and without any indication visible to recipients of the impersonator's calls that delegation has taken place.

initiator: The first principal in a delegation "call chain"; the only participant in the call chain which is not the recipient of a call.

integrity: In security terms, the property that a system always faithfully and effectively enforces all of its stated security policies.

interceptor: An object which provides one or more specialized services, at the ORB invocation boundary, based upon the context of the object request,. The OMG CORBAsecurity specification define the security interceptors.

intermediate: An object in a delegation "call chain" which is neither the initiator nor the ultimate (final) target.

IETF: Internet Engineering Task Force. Reviews an issues Internet standards.

IIOP: Internet Interoperable Object Protocol (specified in the OMG CORBA 2.0/ Interoperability specification).

IOR: Interoperable Object Reference - a data structure specified in the OMG CORBA 2.0/ Interoperability specification.

ITSEC: Information Technology Security Evaluation Criteria (of ECSC-EEC-EAEC). Harmonized Criteria.

MAC: Mandatory Access Control - an access control regime wherein resource access control policy information is always managed by a designated authority, regardless of who creates the resources.

locality constrained: an object is locality constrained if it cannot be accessed from outside a specific locality. references to the object cannot be meaningfully passed outside the boundaries of the locality of concern.

mechanism: A specific implementation of security services, using particular algorithms, data structures, and protocols.

message protection: Security protection applied to a message to protect it against unauthorized access or modification in transit between a client and a target.

mutual authentication: The process whereby each of two communicating principals authenticates the other's identity. Frequently this is a prerequisite for the establishment of a secure association between a client and a target.

Non-Repudiation: The provision of evidence which will prevent a participant in an action from convincingly denying his responsibility for the action.

ORB Core: The functionality provide by the CORBA Object Request Broker which provides the basic representations of objects and the communication of requests.

ORB Services: Elements of functionality provided transparently to applications by the CORBA Object Request Broker in response to the implicit context of an object request.

ORB technology domain: A set of objects or entities that share a common ORB implementation technology.

originator: The entity in an object request which creates the request.

passive threat: The threat of unauthorized disclosure of information without changing the state of the system.

physical security: The measures used to provide physical protection of resources against deliberate and accidental threats.

POSIX: Portable Open System Interfaces (for) UNIX - A set of standardized interfaces to UNIX systems specified by IEEE Standard 1003.

principal: A user or programmatic entity with the ability to use the resources of a system.

privacy: 1. See confidentiality. 2. The right of individuals to control or influence what information related to them may be collected and stored and by whom that information may be disclosed.

private key: In a public-key (asymmetric) cryptosystem, the component of a key pair which is not divulged by its owner.

privilege: A security attribute (q.v.) which need not have the property of uniqueness, and which thus may be shared by many users and other principals. Examples of privileges include groups, roles, and clearances.

proof of delivery: Non-repudiation evidence demonstrating that a message or data has been delivered.

proof of origin: Non-repudiation evidence identifying the originator of a message or data.

proof of receipt: Non-repudiation evidence demonstrating that a message or data has been received by a particular party.

protection boundary: The domain boundary within which security services provide a known level of protection against threats.

PDU: Protocol Data Unit. The data fields of a protocol message, as distinguished from the protocol header and trailer fields.

POA: Portable Object Adaptor

proof of submission: Non-repudiation evidence demonstrating that a message or data has been submitted to a particular principal or service.

public key: In a public-key (asymmetric) cryptosystem, the component of a key pair which is revealed.

public-key cryptosystem: An encryption system which uses an asymmetric-key (q.v.) cryptographic algorithm.

QOP: Quality of Protection. The type and strength of protection provided by a message-protection service.

RPC: Remote Procedure Call.

replaceability: The quality of an implementation which permits substitution of one security service for another semantically similar service.

repudiation: Denial by one of the entities involved in an action of having participated in all or part of the action.

RFP: Request for Proposal. An OMG procedure for soliciting technology from OMG members.

right: A named value conferring the ability to perform actions in a system. Access control policies grant rights to principals (on the basis of their security attributes); in order to make an access control decision, access decision functions compare the rights granted to a principal against the rights required to perform an operation.

rights type: A defined set of rights.

role: A privilege attribute representing the position or function a user represents in seeking security authentication. A given human being may play multiple roles and therefore require multiple role privilege attributes.

RSA: An asymmetric encryption algorithm invented by Ron Rivest, Adi Shamir, and Len Adelman.

seal: To encrypt data for the purpose of providing confidentiality protection.

secret-key cryptosystem: A cryptosystem which uses a symmetric-key (q.v.) cryptographic algorithm.

secure time: A reliable Time service that has not been compromised, and whose messages can be authenticated by their recipients.

security association: The shared security state information which permits secure communication between two entities.

security attributes: Characteristics of a subject (user or principal) which form the basis of the system's policies governing that subject.

security audit: The facility of a secure system which records information about security-relevant events in a tamper-resistant log. Often used to facilitate an independent review and examination of system records and activities in order to test for adequacy of system controls, to ensure compliance with established policy and operational procedures, to detect breaches in security, and to recommend changes in control, policy and procedures.

security features: Operational information which controls the security protection applied to requests and responses in a CORBA Security conformant system.

security context: The CORBA Security object which encapsulates the shared state information representing a security association.

security policy: The data which defines what protection a system's security services must provide. There are many kinds of security policy, including access control policy, audit policy, message protection policy, non-repudiation policy, etc.

security policy domain: A domain whose objects are all governed by the same security policy. There are several types of security policy domain, including access control policy domains and audit policy domains.

security service: Code that implements a defined set of security functionality. Security services include Access Control, Audit, Non-repudiation, and others.

security technology domain: A set of objects or entities whose security services are all implemented using the same technology.

subject: An active entity in the system; either a human user principal or a programmatic principal.

symmetric key: The key used in a symmetric ("secret-key") encryption system. In such systems, the same key is used for encryption and decryption.

tagged profile: The data element in an IOR which provides the profile information for each protocol supported.

target: The final recipient in a delegation "call chain." The only participant in such a call chain which is not the originator of a call.

target ACI: The Access Control Information for the target object.

target object: The recipient of a CORBA request message.

threat: A potential violation of security.

traced delegation: Delegation wherein information about the initiator and all intervening intermediates is available to each recipient in the call chain, or to the authorization subsystem controlling access to each recipient.

trust model: A description of which components of the system and which entities outside the system must be trusted, and what they must be trusted for, if the system is to remain secure.

trusted code: Code assumed to always perform some specified set of operations correctly.

TCB: Trusted Computing Base. The portion of a system which must function correctly in order for the system to remain secure. A TCB should be tamper-proof and its enforcement of policy should be noncircumventable. Ideally a system’s TCB should also be as small as possible, to facilitate analysis of its integrity.

TCSEC: Trusted Computer System Evaluation Criteria (a U.S. Department of Defense Standard specifying requirements for secure systems).

unauthenticated principal: A user or other principal who has not authenticated any identity or privilege.

UNO: Universal Networked Objects (an OMG Specification, now obsolete).

UTC: Coordinated Universal Time.

unsecure time: Time obtained from an unsecure time services.

UTO: Universal Time Object (c.f. CORBA Time Service [3]).

user: A human being using the system to issue requests to objects in order to get them to perform functions in the system on his behalf.

user sponsor: The interactive user interface to the system which acts as the authenticating authority (e.g. validating passwords) which validate the identity of a user.

vault: The CORBA Security object which creates security context objects.

X/Open: X/Open Company Ltd., U.K.