Security Service Specification 15

15.1 Introduction to Security

15.1.1 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.

15.1.2 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 respond to 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.

15.1.3 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 E, 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.

15.1.4 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.

15.1.5 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.

Simplicity

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

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).

Scalability

It should be possible to provide security for a range of systems from small, local systems to large intra- and interenterprise 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.

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.

Usability of 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.

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.

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 pssible 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.

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.

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, for example, 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.

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.

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.

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.

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 between countries; 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 or ITSEC 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.

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.

15.2 Introduction to the Specification

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. It includes all the security facilities required in the OS RFP3 and associated OMG White Paper on Security, except where it is felt that this would be too big a step at this stage (particularly when relevant standards are not in place). The specification includes:

- 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.

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 have been identified as a Common Facility. 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. Additional security facilities could be designed as ORB extensions, Security Object services, or Common Facilities, as appropriate.

15.2.1 Conformance to CORBA Security

Conformance to CORBA security covers:

- **Main security functionality**. There are two possible levels:
  - *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.
- **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 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.

  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 Policies.
- **Security Replaceability**. This 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 15.7, Implementor's Security Interfaces. 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.

If the ORB does not conform to one of these replaceability options, the standard security policies defined in this specification cannot be replaced by others, nor can the implementation of the Security services. For example, it would not be possible to replace the standard access policy by a label-based policy if one of the replaceability options 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.

An ORB that supports one or both of these replaceability options may be *Security Ready* (i.e., supports no security functionality itself, but ready to have security added) or may support security functionality Level 1 or Level 2.

- **Secure Interoperability**. Possibilities are:
  - *Secure Interoperability - Standard:* An ORB conforming to standard secure interoperability 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 15.8, Security and Interoperability, if they both use the same underlying security technology.
  - *Standard plus DCE-CIOP Option:* An ORB conforming to standard plus DCE-CIOP secure interoperability supports all functionality required by standard secure interoperability, and also provides secure interoperability (using the DCE Security services) between ORBs using the DCE-CIOP protocol.

  If the ORB does not conform to one of these, it does not use the GIOP security enhancements, so will interoperate securely only in an environment-specific way.

The conformance statement required for a CORBA conformant security implementation is defined in Appendix F, Conformance Statement. This includes a table which can be ticked to show what the ORB conforms to.

15.2.2 Specification Structure

Normative and Non-normative Material

This specification contains normative and non-normative (explanatory) material. Only Sections 15.5 through 5.8 and Appendices A, B, D, and F are normative.

Section Summaries

Section 15.1 and its subsections, which is an introduction to security, explains why security is needed in distributed object systems, and enumerates the security requirements for secure distributed object systems.

Section 15.2 and its subsections provide an introduction to and overview of the specification.

Section 15.3 and its subsections describe the **security reference model**, which provides the overall framework for CORBA security.

Section 15.4 and its subsections describe the **security architecture**, which underlies this specification. This introduces different users' views of security and gives an outline of how secure CORBA-compliant systems are constructed. It also presents high level models of the objects involved for different views, and describes how they are used.

Section 15.5 and its subsections specify the security facilities and interfaces available to **application developers**. Most functions can be implemented transparently to application, though interfaces and additional functionality are available to security-aware applications.

Section 15.6 and its subsections specify the **administrator's** facilities and interfaces. Only essential administration functions are defined by this specification; other administrative capabilities are expected to be developed outside the Object Services Program.

Section 15.7 and its subsections specify the **Implementors interfaces** used to build secure CORBA systems. This section specifies the IDL interfaces of the security objects available to ORB implementors, and describes the relationship and dependencies of these objects on the ORB core and also on external Security services, where these are used.

Section 15.8 and its subsections specify the architecture for **interoperability** in a secure, distributed object system. It also specifies how security affects the CORBA 2 GIOP/IIOP and DCE ESIOP interoperability protocols.

Appendix A, Consolidated OMG IDL, contains the complete OMG IDL specification, including the module structure, of the interfaces defined in this document, except for those that are CORBA core extensions and defined in Appendix B, Summary of CORBA 2 Core Changes.

Appendix B, Summary of CORBA 2 Core Changes, describes the changes required to the CORBA 2 core for security.

Appendix C, Relationship to Other Services, describes the relationship of the Security services to other object services and to the common facilities.

Appendix D, Conformance Details, describes in more detail what conformance to the security functionality conformance levels and the security implementation conformance points requires.

Appendix E, Guidelines for a Trustworthy System, provides guidelines for implementation of a trustworthy system, which provides protection against the security threats in a distributed object system with the required assurance of its correctness and effectiveness.

Appendix F, Conformance Statement, describes the conformance statement, which must accompany a secure CORBA implementation and what this implementation must contain.

Appendix G, Facilities Not in This Specification, outlines security facilities that have not been included in this specification, but left for another phase of security specifications.

Appendix H, Interoperability Guidelines, includes guidelines for defining security mechanism tags in interoperable object references, and examples of the use of the secure inter-ORB protocol SECIOP.

Appendix I, Glossary.

Proof of Concept

With the exception of Audit, Non-repudiation services, and the revised IIOP protocol extensions for security, all the facilities in this specification have been prototyped by at least one of the submitting companies.

The Non-repudiation Service interfaces are based upon the draft IETF Non-repudiation functionality as defined in the IDUP-GSS-API proposal.

15.3 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.

15.3.1 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 15-1 depicts the model for CORBA secure object systems. All object invocations are mediated by appropriate security to enforce policies such as access controls. These functions should be tamper-proof, always be invoked when required by security policy, and function correctly.

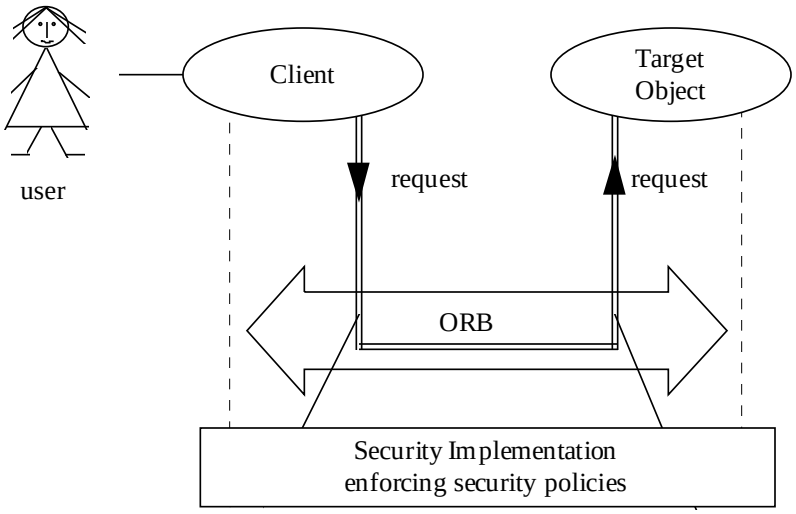

Figure 15-1 A Security model for object systems

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 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 building 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.

15.3.2 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 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 Access Policies under Section 15.3.4). 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:

- Some attributes may be available, without authentication, to any principal. This specification defines one such attribute, called *Public*.
- Some attributes are acquired through authentication; identity attributes and privilege attributes are in this category.
- 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 15-2.

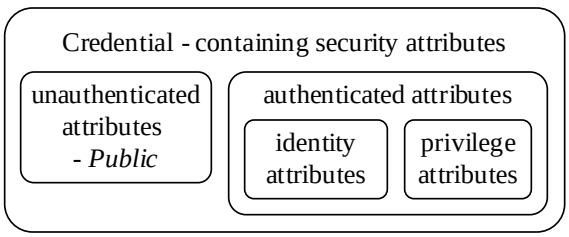

Figure 15-2 Credential containing security attributes

15.3.3 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 15-3.

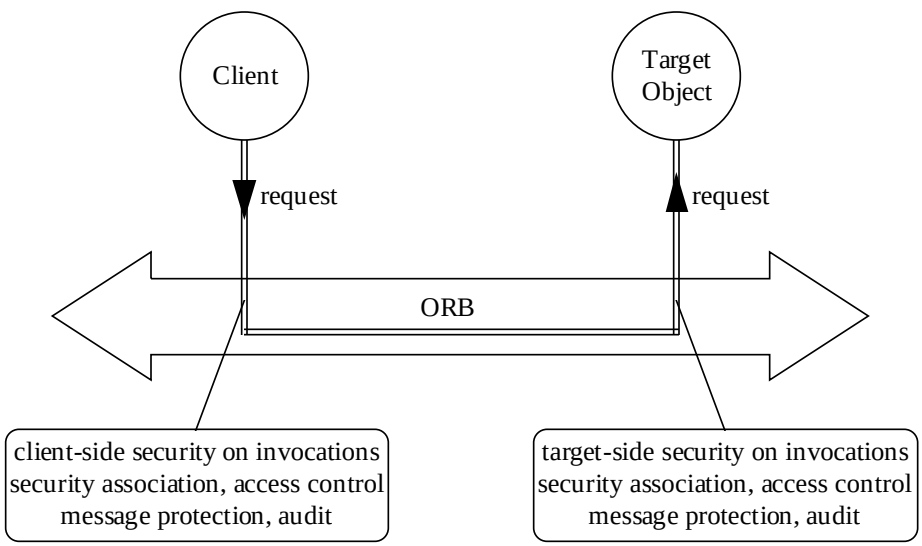

Figure 15-3 Target Object via ORB

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 15.3.4, Access Control Module.
- Auditing this invocation if required, as described in Section 15.3.5, Auditing.
- 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.

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.

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 the following diagram. 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.

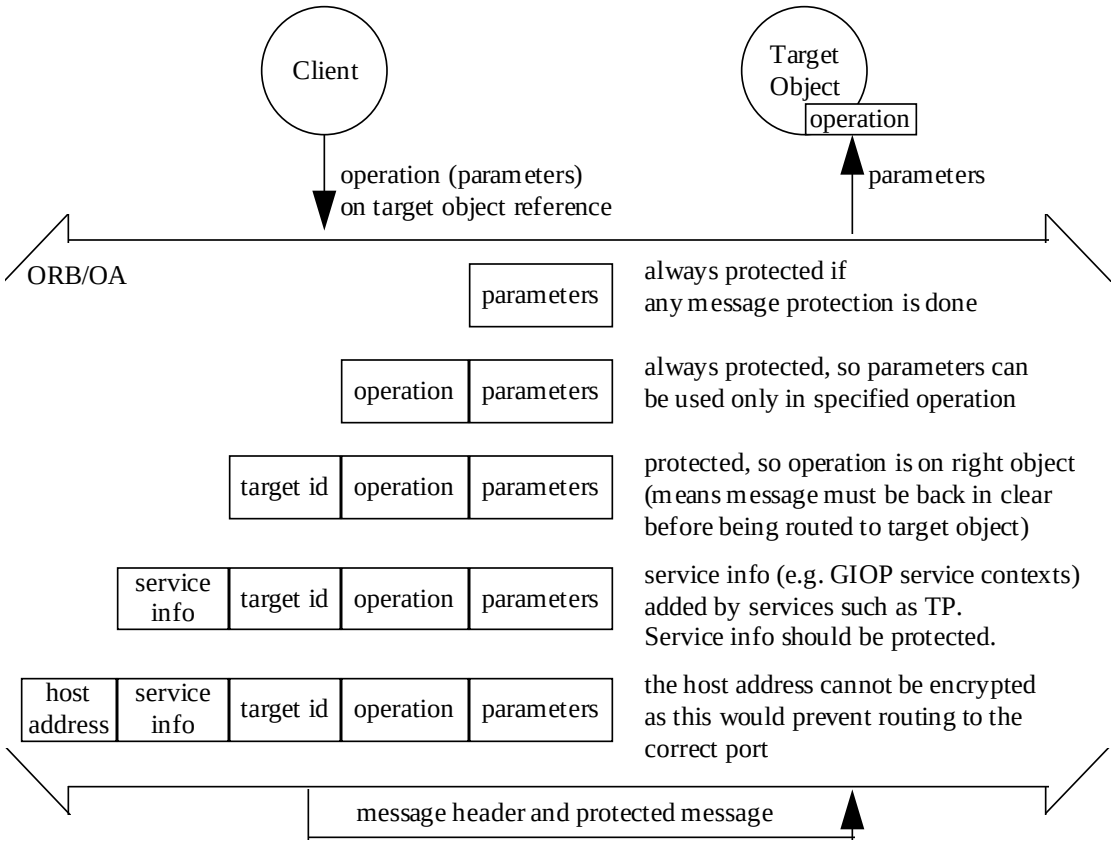

Figure 15-4 Message protection

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 also 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 reprotect the message.

15.3.4 Access Control Model

The model depicted in Figure 15-5 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.

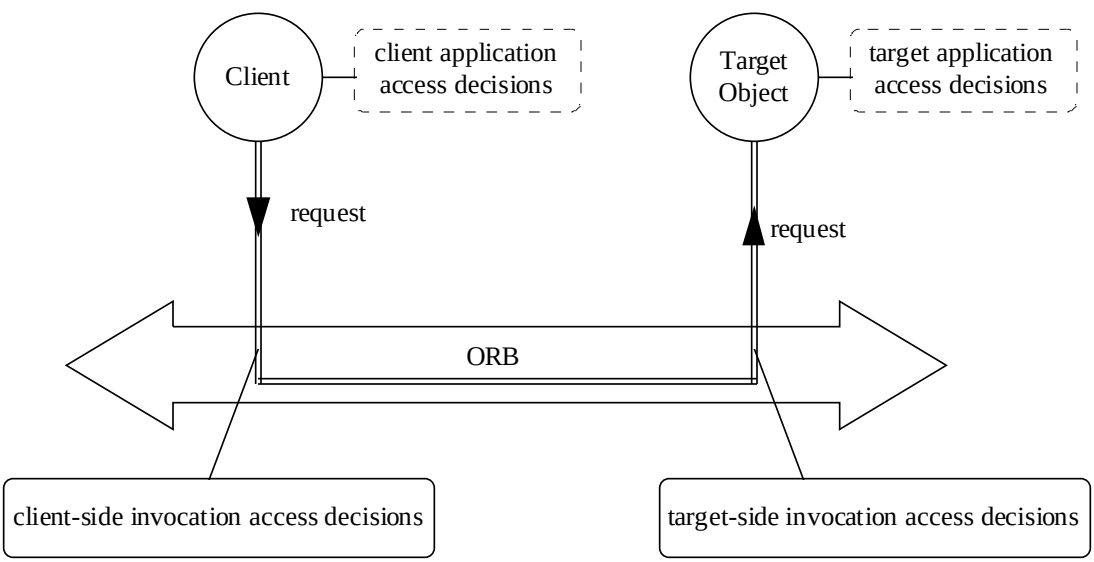

Figure 15-5 Access control model

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.

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 the Access Policies section.

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 15.3.2, Principles and Their Security Attributes). 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 the Access Policies section).

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).

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.

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.

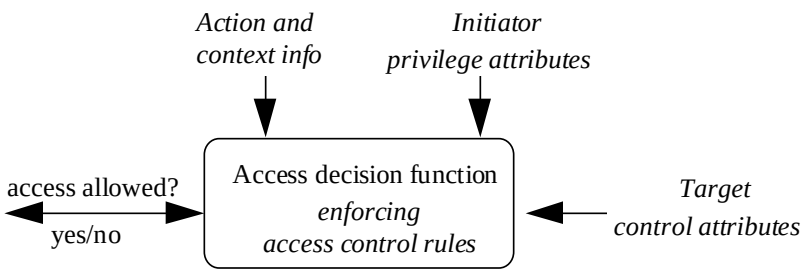

Figure 15-6 Authorization model

The privilege and control attributes are the main variables used to control access, and so this section focuses on these.

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*.

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.

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 15.6.4, Access Policies.

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.

15.3.5 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.

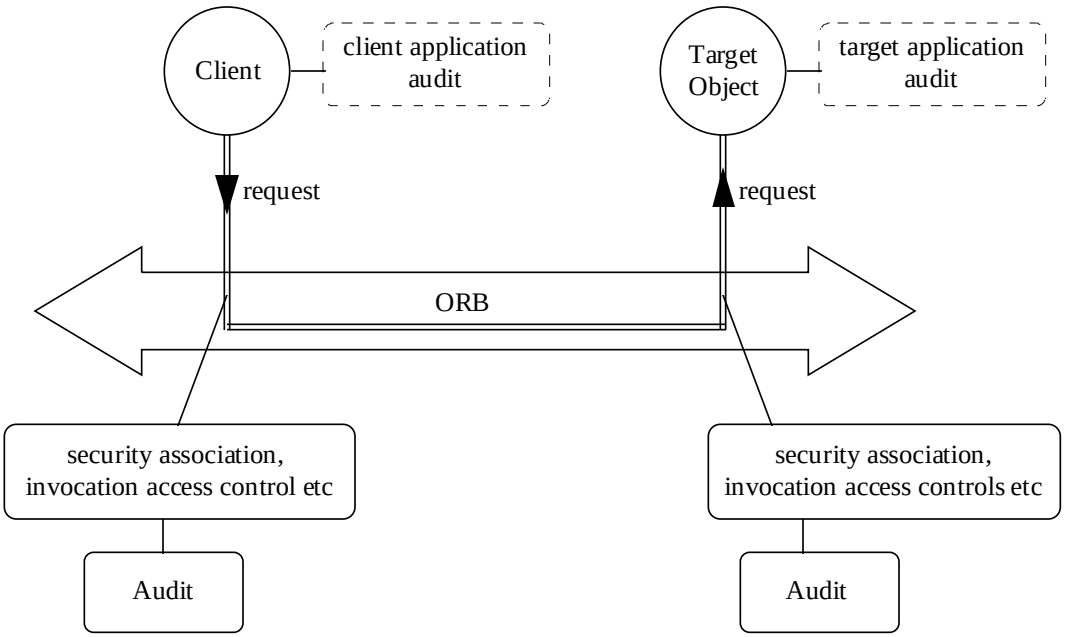

Figure 15-7 Auditing model

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 (failures only 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 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.

15.3.6 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 15-8.

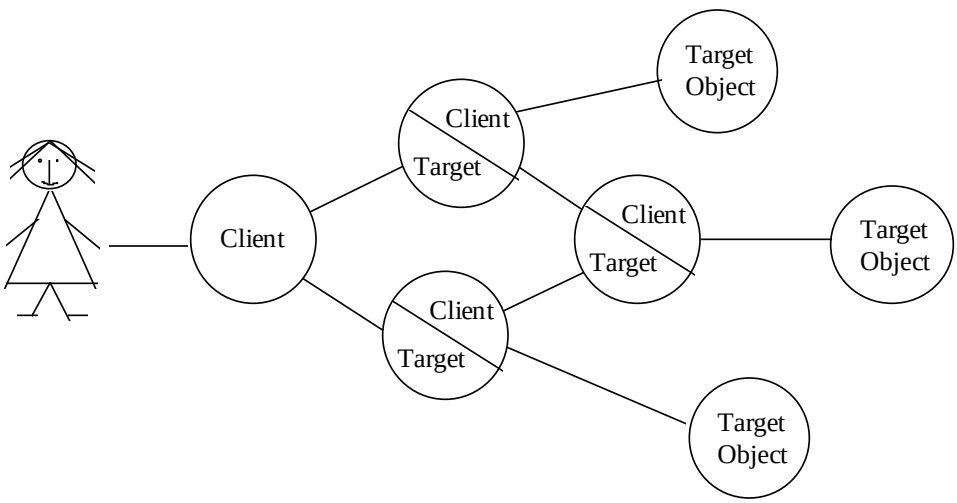

Figure 15-8 Delegation model

This complicates the access model described in Section 15.3.4, Access Control Model, 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 G, 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.

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).

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)

   using
   - (no privileges
   - (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.

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.

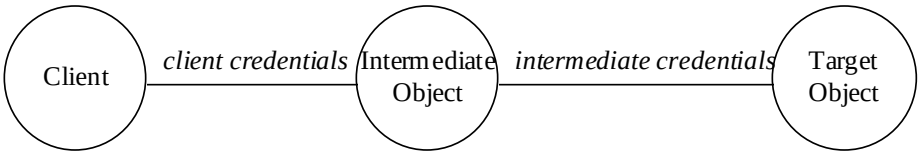

Figure 15-9 No delegation

  - **simple delegation**: the client permits the intermediate to assume its privileges, both using them for access control decisions and delegating them to other 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).

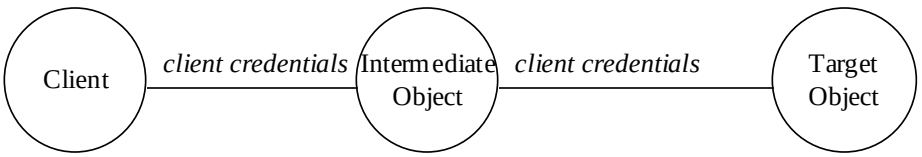

Figure 15-10 Simple delegation

- **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.

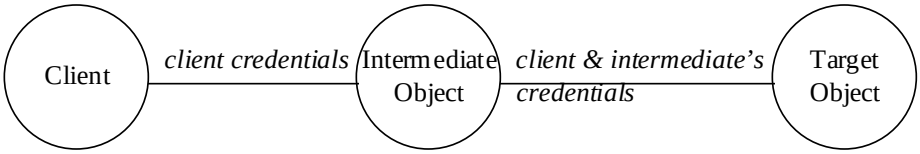

Figure 15-11 Composite delegation

- **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.

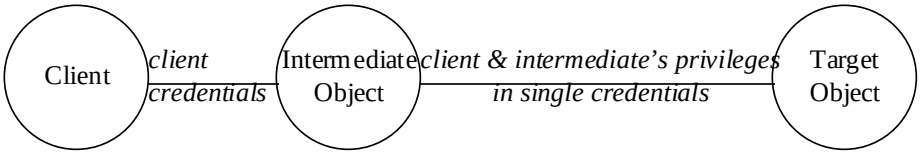

Figure 15-12 Combined privileges delegation

- **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.

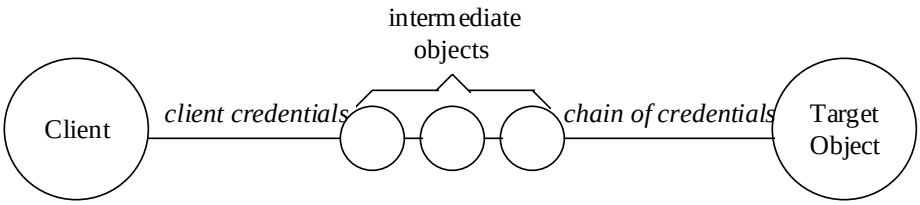

Figure 15-13 Traced delegation

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.

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.

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 *NotSupported* exception returned. Note that these interfaces do not allow the application to set controls such as target restrictions. Appendix G, Facilities Not in This Specification, includes potential future advanced delegation facilities, which include such controls.

15.3.7 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:

- The non-repudiation policy (or policies) applicable to the evidence,
- The type of action or event,
- The 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 15-14.

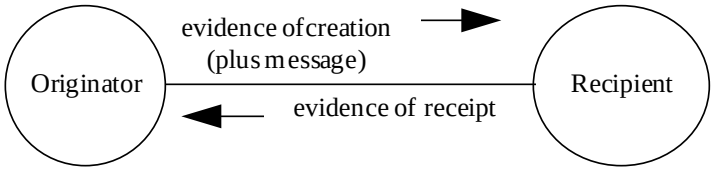

Figure 15-14 Proof of receipt

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.

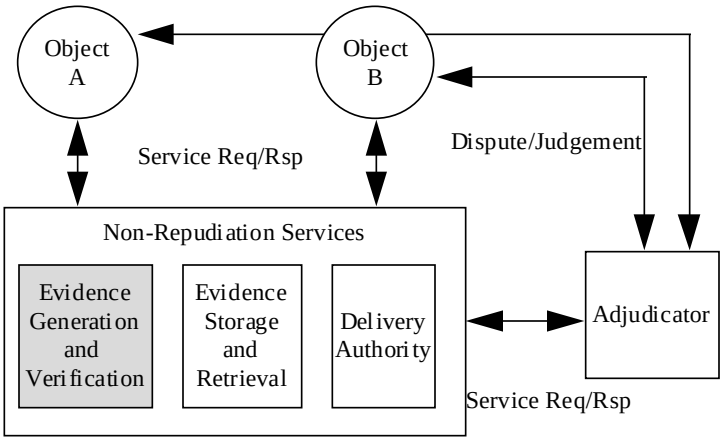

Figure 15-15 Non-repudiation services

The non-repudiation services illustrated in Figure 15-15 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.

15.3.8 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.

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.

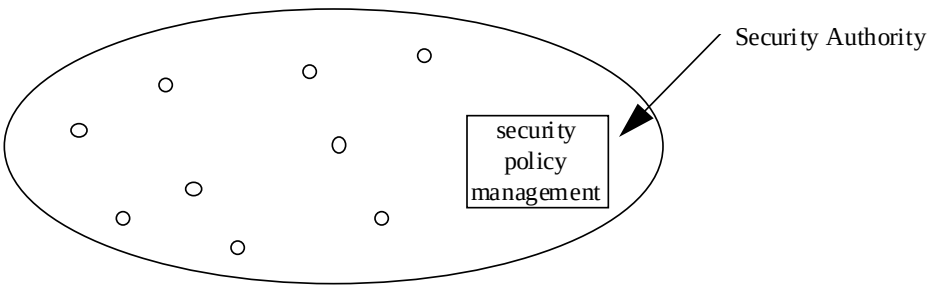

Figure 15-16 Security policy domains

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.

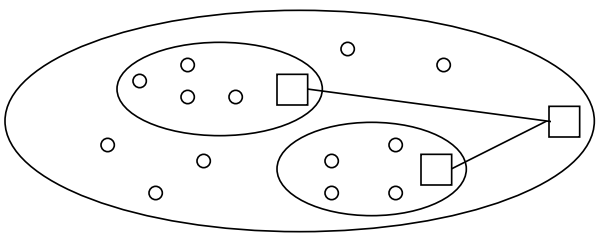

Figure 15-17 Policy domain hierarchies

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.

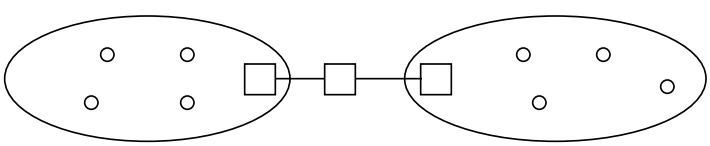

Figure 15-18 Federated policy domains

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.

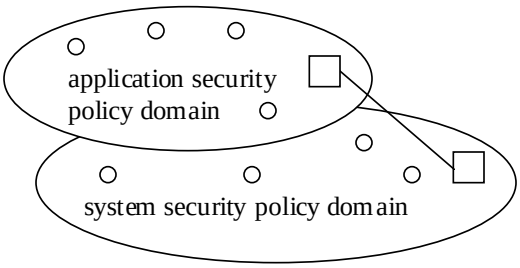

Figure 15-19 System- and application-enforced policies

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.

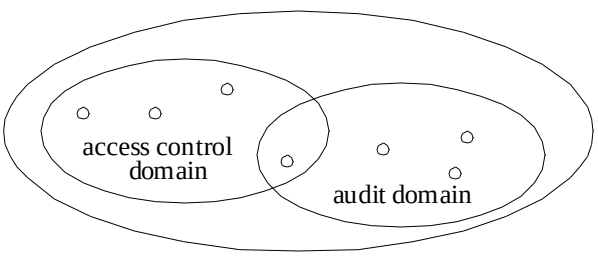

Figure 15-20 Overlapping policy domains

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).

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:

- **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
- **Identity domains**. Objects in an identity domain can share the same identity. Objects in the same identity domain and
  - 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).

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 can be implemented using a wide variety of security mechanisms and protocols.

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 first 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.

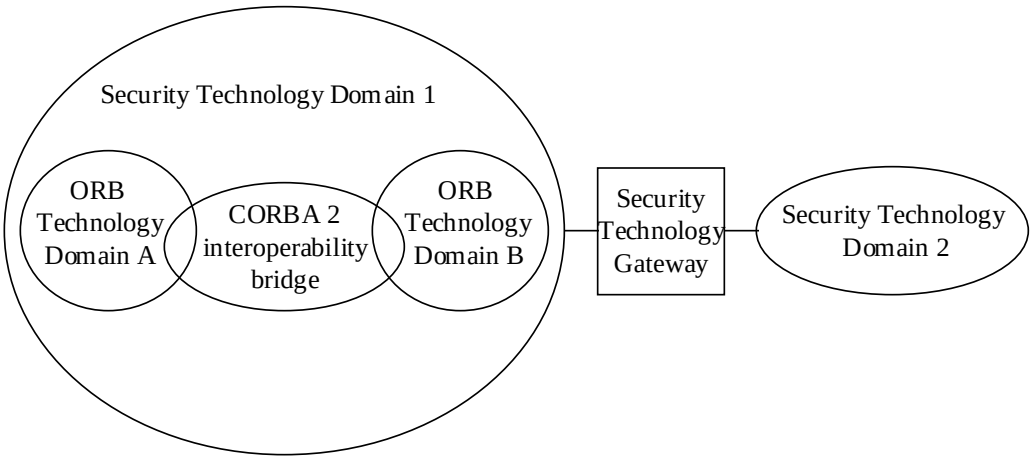

Figure 15-21 Framework of domains

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. (This specification extends the CORBA 2 interoperability specification to detail how security fits in it.) 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.

15.3.9 Security Management and Administration

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

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 and also affected by the Collections Service, 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 interfaces 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.

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, so no management interfaces for it are specified here.

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.

15.3.10 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 F, Conformance Statement.)

15.4 Security Architecture

This section explains how the security model is implemented. It describes the complete architecture as needed to support all conformance levels described in Section 15.2.1, Conformance to CORBA Security. Not all of these levels are mandatory for all implementors to support.

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.

15.4.1 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

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.

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 15.3, Security Reference Model.

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 15.5, Application Developer’s Interfaces.

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. It is expected that the CORBA Common Facilities will, in the future, include management facilities that can manage security, as well as other objects, in an OMA-compliant system. 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 15.6, Administrator's Interfaces.

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 uses ORB Security services during object invocation, as defined in Section 15.4.2, Structural Model. 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.

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 15.7, Implementor's Security Interfaces. The implications of constructing the system securely to meet threats are described in Appendix E, Guidelines for a Trustworthy System.

15.4.2 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:

- Application-level components, which may or may not be aware of security;
- 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.
- Components implementing specific security technology;
- Basic protection and communication, generally provided by a combination of hardware and operating system mechanisms.

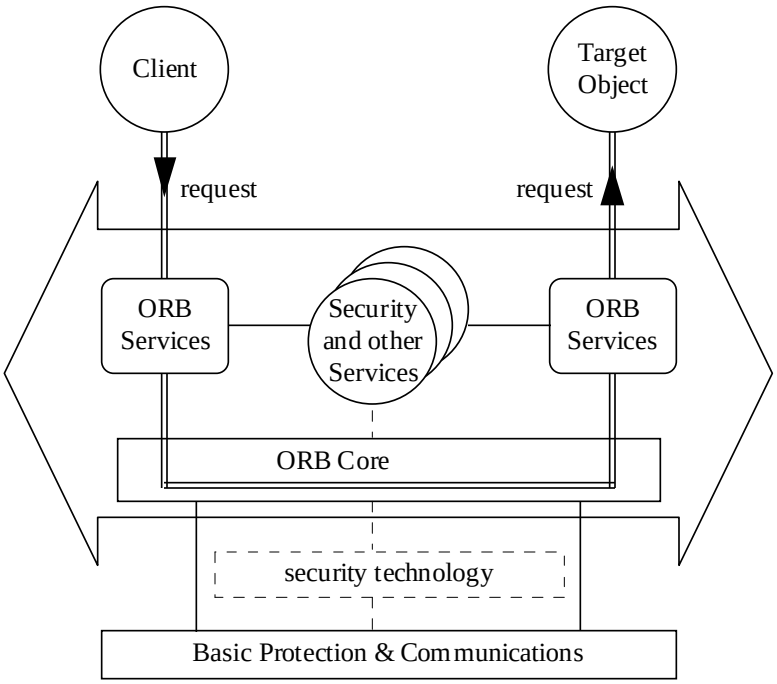

Figure 15-22 Structural model

Figure 15-22 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.

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 15.4.5, Security Object Models). As in the Object Management Architecture, the client may, or may not, be an object.

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 (some of) 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.

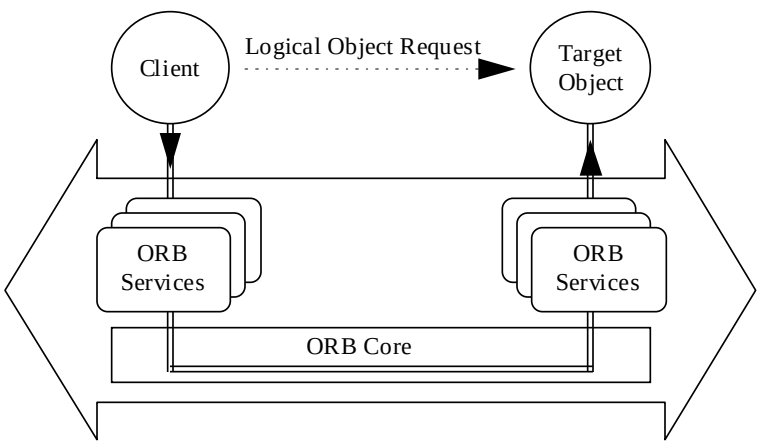

Figure 15-23 ORB services

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

The Security Architecture builds upon the CORBA 2 Interoperability Architecture in considering the selection of ORB Services as part of the process of establishing a **binding** between a client and a target object.

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, an object reference defines those policies and static properties of the target object that affect how the client's ORB establishes a binding to the object, for example, the quality of protection needed. Subsequently there may be a need to modify or extend details of the binding for a particular invocation (e.g. when a request is required to be transactional).

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

- The object reference of the target object.
- Elements of client context, for this binding, associated with particular ORB or Object Services (e.g. access policy domain, security context).

A binding is distinct from the target object to which it is made, though uniquely associated with it. The state associated with a binding is accessible via operations on the target object reference on the client side (which are completely disjointed from its application level operations), and via a "Current" object at the target side.

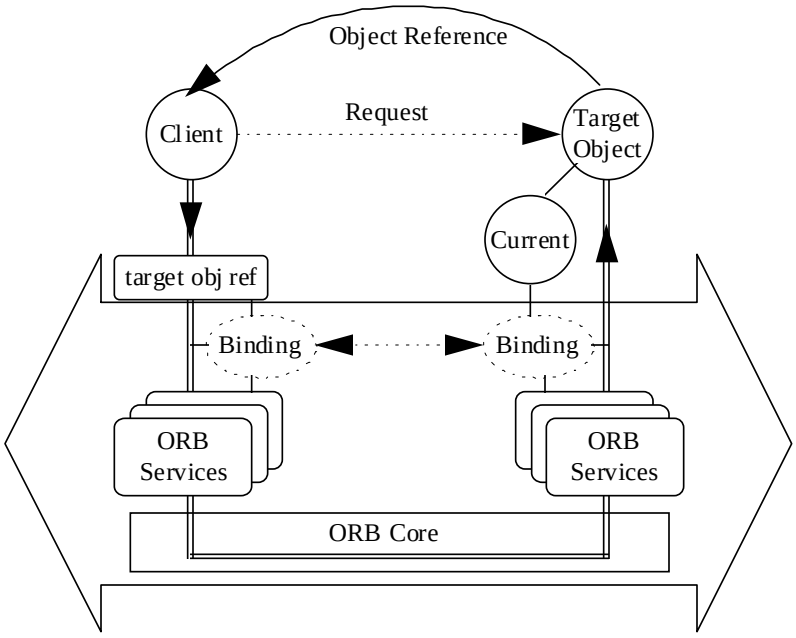

Figure 15-24 Object reference

If a client requires to establish several distinct, independent bindings to the same target object, then it can make a copy of an existing object reference. Any binding established via the new reference is distinct from bindings used with the old reference.

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 Object Security Services for authentication, access control, audit, non-repudiation, and secure invocations. These security services form the Security Replaceability Conformance option.

These object security services may in turn call on external security services to implement security technology.

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 15.3.8, Domains. 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.

On object creation, the ORB implicitly associates the object with one or more Security Policy domains as described in Administrative Model, under Section 15.4.5, Security Object Models. An implementation may allow objects to be moved between domains later.

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 Administrative Model, under Section 15.4.5, 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.)

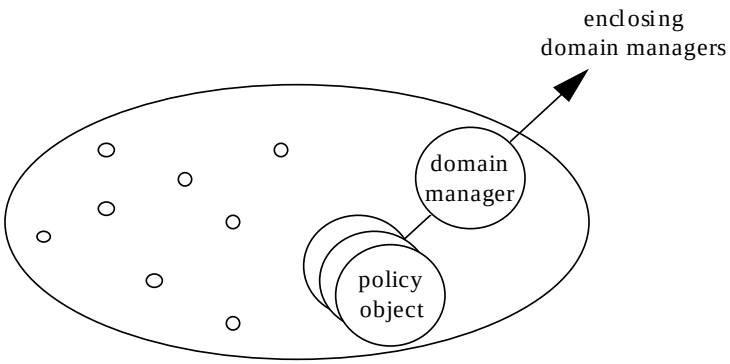

Figure 15-25 Domain objects

Where an object is a member of more than one domain, for example, there is a hierarchy of domains, the object 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 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 the subsections The Model as Seen by Applications, and The Model as Seen by the Objects, under Section 15.4.5, Security Object Models). 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 Administrative Model, under Section 15.4.5.)

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 The Model as Seen by Applications, under Section 15.4.5.)

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 such as Management Facilities and/or Collection Service in the future.

15.4.3 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.

15.4.4 Basic Protection and Communications

Environment Domains

As described in Security Environment Domains, under Section 15.3.8, Domains, 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.

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.

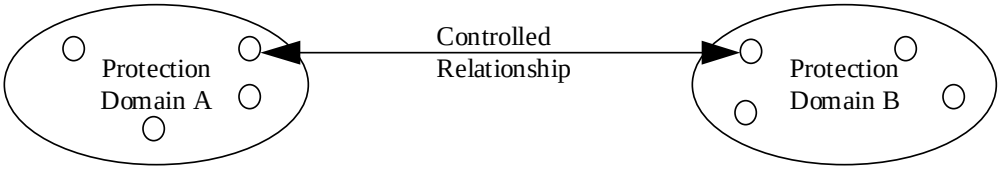

Figure 15-26 Controlled relationship

In this architecture, the trusted components supporting security services are encapsulated by objects, as described in The Model as Seen by the Objects Implementing Security, under Section 15.4.5, Security Object Models. Clearly, objects that encapsulate sensitive security information must be protected to ensure that they can only be accessed in an appropriate way.

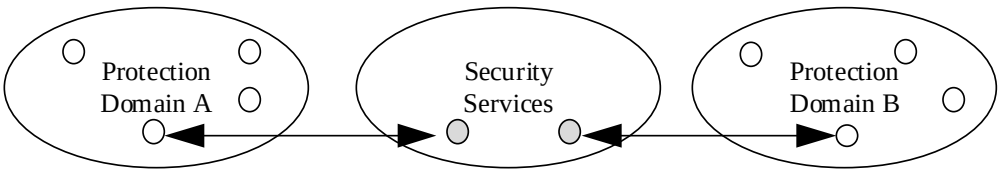

Figure 15-27 Object encapsulation

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.

15.4.5 Security Object Models

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

- The model as seen by applications.
- The model as seen by administrators and administrative applications.
- 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.

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.

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, needs this).

Establishing Credentials

If the principal has already been authenticated outside the object system, then Credentials can be obtained from Current (see later).

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 15-28.

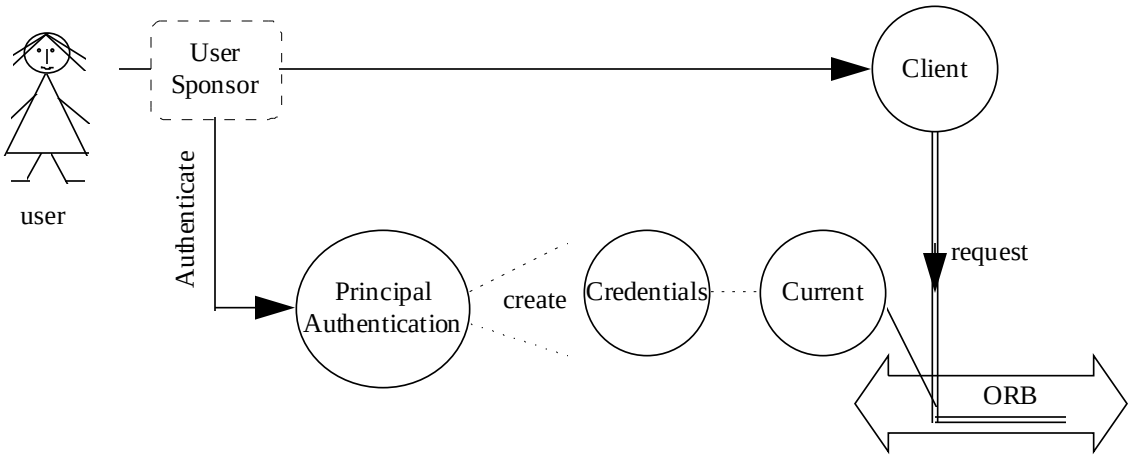

Figure 15-28 Authentication

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 on the Principal Authenticator (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 (thread of activity) at both client and target objects. In a secure environment, the Current object supports the SecureCurrent interface, which gives access to security information. Current retains a reference to the Credentials associated with the execution environment. Object invocations use Credentials in Current. If a user sponsor is used, it should set the user's credentials as the default 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 available via Current.

Handling Multiple Credentials

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

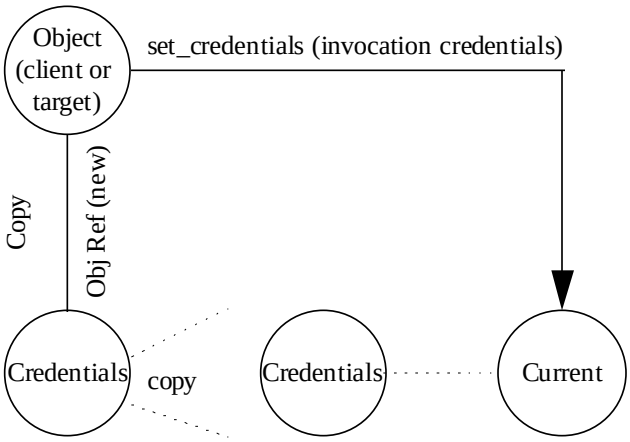

Figure 15-29 Multiple credentials

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

When all required changes have been made, the **`set_credentials`** operation can be used on the Current object 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 **`get_credentials`** on Current, asking for the invocation credentials.

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 of changing security attributes for a principal:

- 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.
- Setting attributes on the target object reference (meaning on the binding as described in ORB Services, under Section 15.4.2, Structural Model). Attributes set here 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.

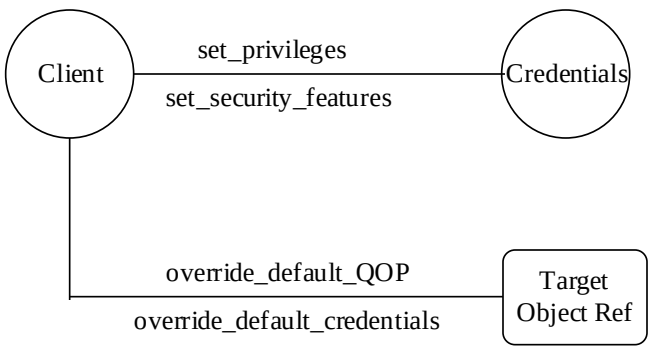

Figure 15-30 Changing security attributes

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.
- **`set_security_features`** to set such features as the quality of protection of messages (and the credentials to use for future invocations when at an intermediate object).

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 Section G.9, Target Control of Message Protection).

A client may want to use different privileges or controls when invoking different targets. It can do this by using **`override_default_credentials`** specifying the credentials to be used with that target. 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 **`override_default_QOP`**, specifying a QOP on the target object reference and then resetting this QOP when confidentiality is no longer required.

From the application's point of view, the **`override_default`** operations are normal invocations. However, they are actually operations upon the reference to the target object rather than the target object itself.

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) and **`get_security_features`** on credentials objects and **`get_active_credentials`** and **`get_security_features`** on target objects.

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.

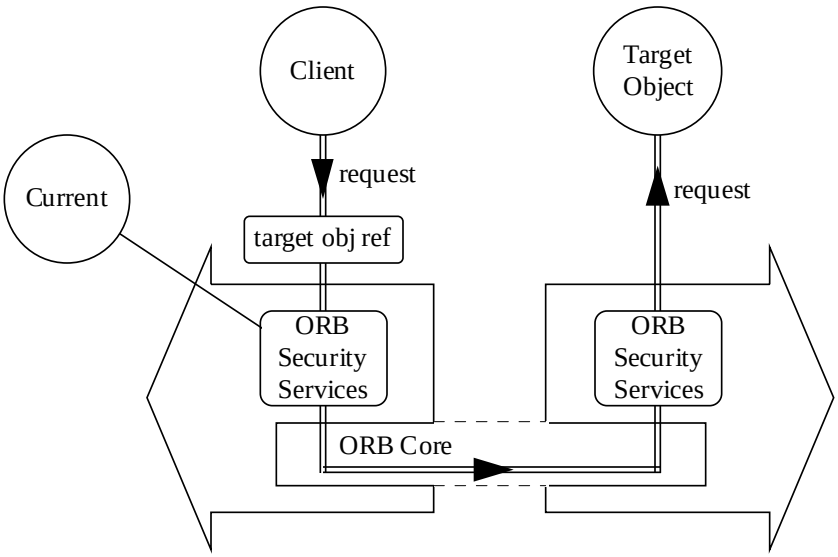

Figure 15-31 Making a secure invocation

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.

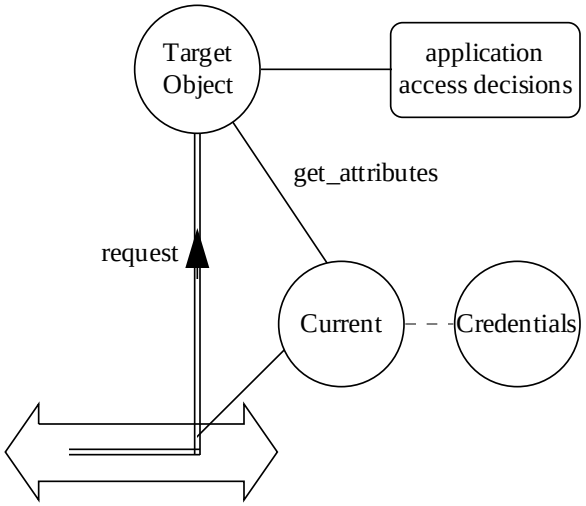

Figure 15-32 Target object security

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

The **`get_attributes`** operation can also be used at the client and can be used on any Credentials object, not just on Current. When called on Current, it always gets the incoming credentials from the client at the target object, and from the user at the client machine.

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 15-33.

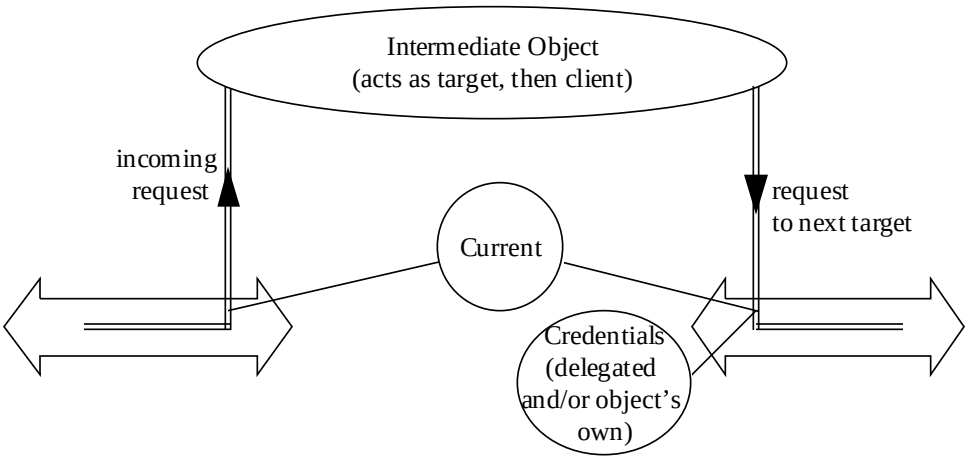

Figure 15-33 Security-unaware intermediate object

For a security-unaware intermediate object, Current retains 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).

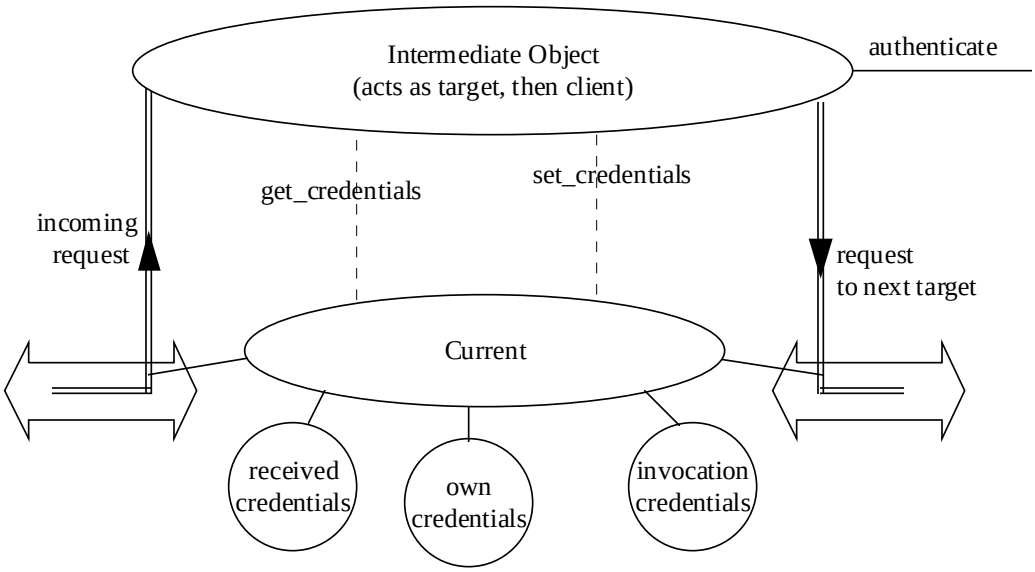

Figure 15-34 Security-aware intermediate object

After a chain of object calls, the target can call **`get_attributes`** on Current 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 on Current to get the incoming credentials. This may be a list of one or more credential objects depending on the authentication and delegation technology used. If more than one credential is returned, the first credential is that of the initiator. Other credentials are of intermediates in the chain. After composite delegation (see Section 15.3.6, Delegation), 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), and then use **`set_privileges`** to modify them. Finally it uses a call to **`set_credentials`** to make the received credentials the default ones for subsequent invocations. When the **`received_credentials`** are passed to **`set_credentials`**, logic under the Current interface determines that a delegation operation is required and does what is necessary transparently.

If the intermediate object wishes to change the association security defaults (for example, the quality of protection) for subsequent invocations, it can do so by using the Current interfaces (e.g. **`override_default_qop`**).

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 to obtain the appropriate credential, and then call **`set_privileges`** to establish the appropriate rights. After doing this, it can use **`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 call **`current::get_credentials`** (specifying own credentials) to get a copy of the credentials (which will have been set up automatically). It can then copy and **`set_privileges`**, etc. on these, as appropriate for the objects it intends invoking.

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 using **`get_security_mechanism`** and **`override_default_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, the application is recommended to 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 15-35.

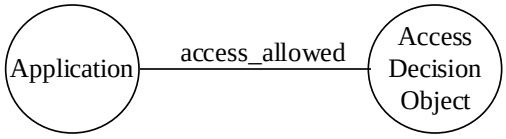

Figure 15-35 access_allowed application

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 Administrative Model, in Section 15.4.5). In this case, the application obtains the Access Decision object as shown in Figure 15-36.

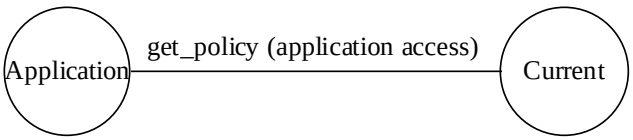

Figure 15-36 get_policy application

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 15.6.5, Audit Policies, 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 sent audit records to different audit trails.

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

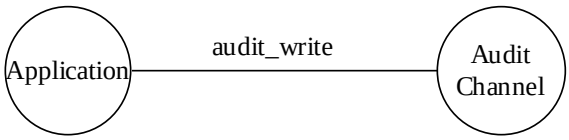

Figure 15-37 audit_write application

If an application is using an audit policy administered via domains, it uses an Audit Decision object (see the Access Decision object) to decide whether to audit an event. It can find the appropriate Audit Decision object using the **`get_policy`** operation on Current as follows.

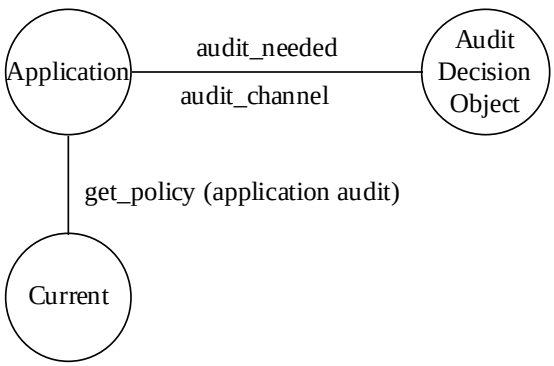

Figure 15-38 Audit decision object

The application invokes the **`audit_needed`** operation on the Audit Decision object, passing the values the Audit Decision object requires 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 15.6.5, Audit Policies.) The Audit Decision object responds with whether an audit record needs to be written to the audit channel or not.

An audit channel can also be associated with an audit policy object, so 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 to find the Audit Channel object to use. However, applications can create their own channel objects.

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 **`get_policy`** on Current, and then the appropriate **`get_`** operation on the policy object obtained as defined in Section 15.6, Administrator's Interfaces (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 a **`set_NR_features`** operation on the NRCredentials object (see the **`set_security_features`** operations on Credentials).

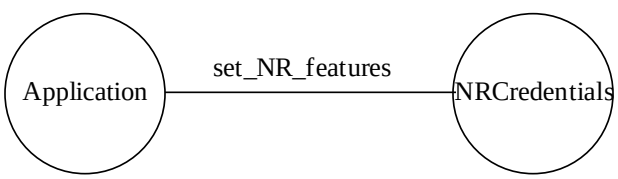

Figure 15-39 set_NR_features operation

`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 **`get_policy`** specifying NRpolicy on Current. 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 a **`get_NR_features`** operation on NRCredentials 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 on an NRCredentials object.

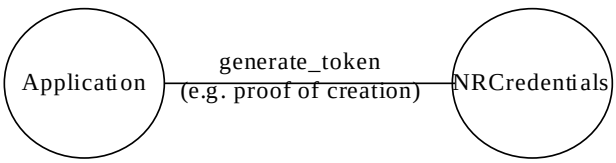

Figure 15-40 generate_token operation

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 15-41 illustrates the policy considerations relating to generation of complete evidence, and the sequence of actions involved in generating and using complete evidence.

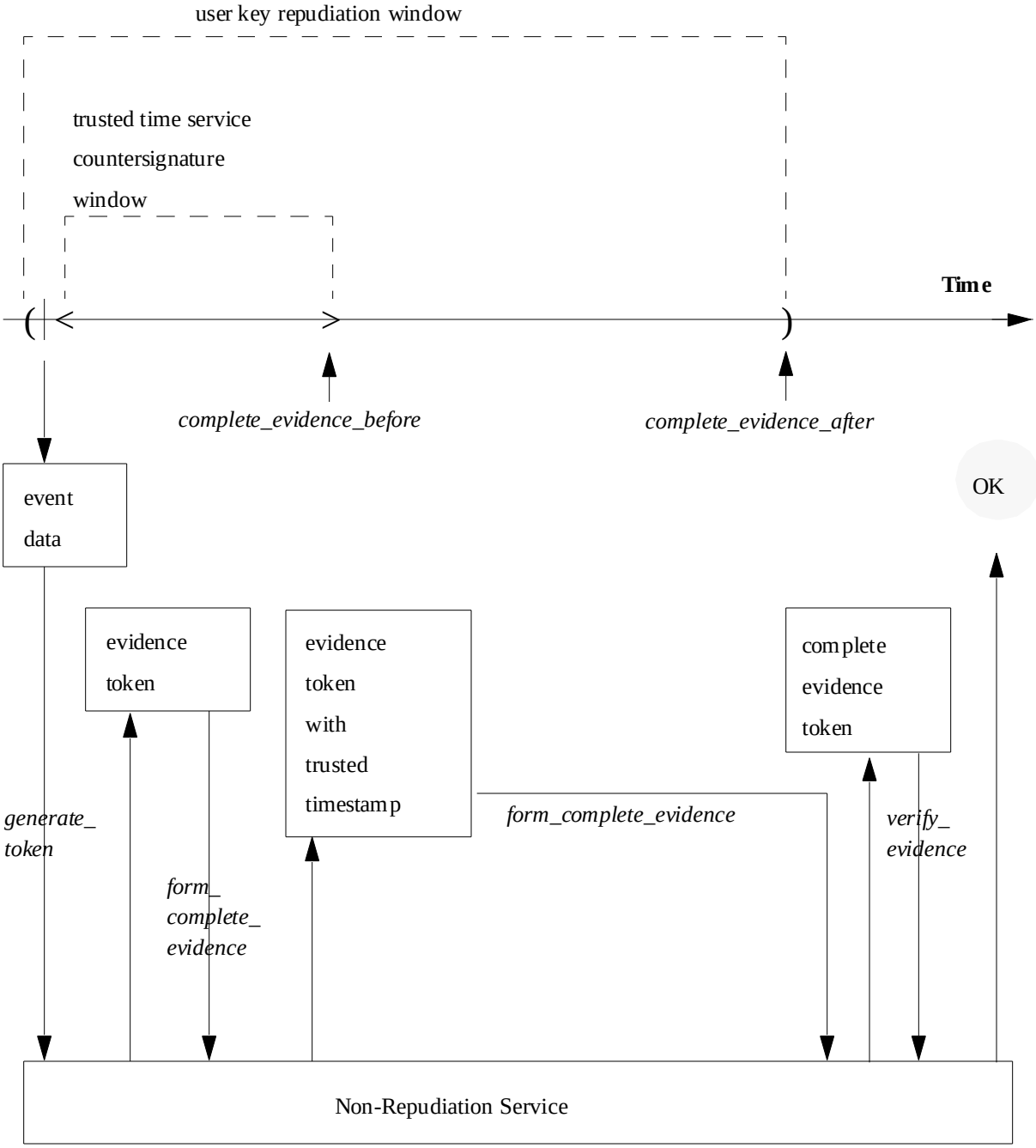

Figure 15-41 Non-repudiation service

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.

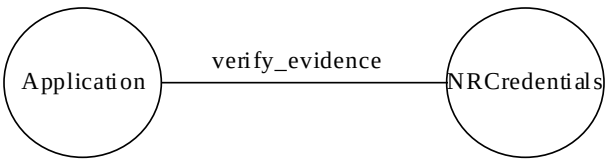

Figure 15-42 verify_evidence operation

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 15-43.

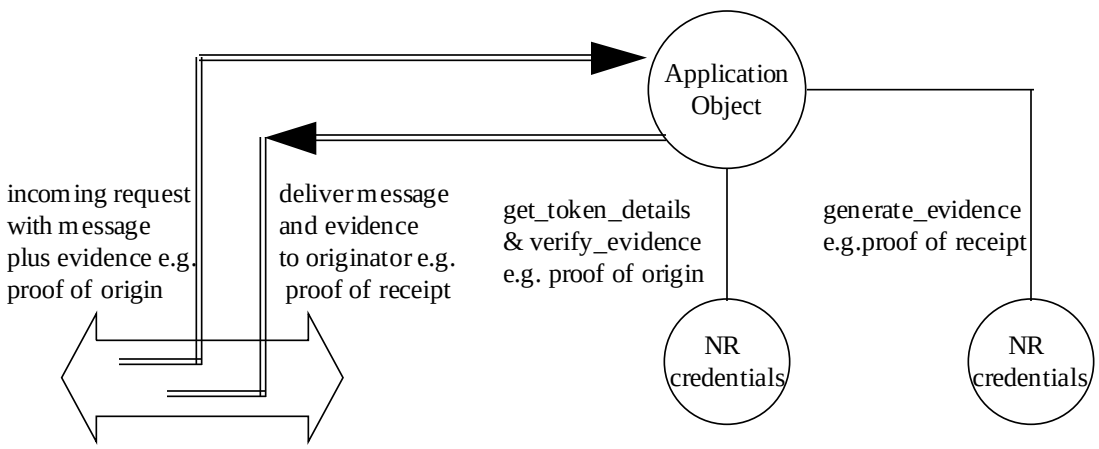

Figure 15-43 Proof of origin message

- 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 **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 **`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 **`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 **`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 **`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 **`verify_evidence`** operation on the NR token, which must contain complete evidence to settle disputes.

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

When a new object is created in a secure environment, the ORB implicitly associates the 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 an object's creation does not need to be aware of security to protect the objects it creates. Also, automatically making an 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 creating object calls **`CORBA::BOA::create`** 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.

Also, in some cases, when a new object 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 is to be unaware of security, the domain manager must be created transparently to the application. A construction policy specifies whether new objects 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 **`BOA::create`** or equivalent.

Other Domain and Policy Administration

Once an object 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 document).

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.

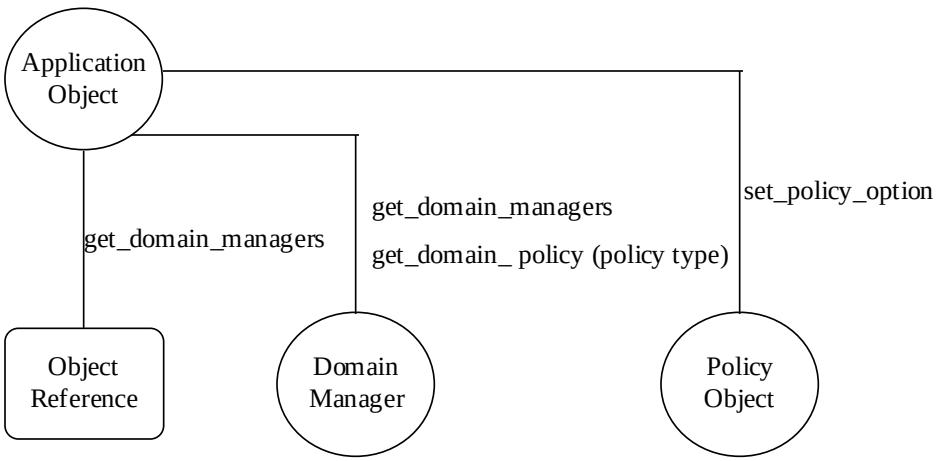

Figure 15-44 Managing security policies

Finding Domain Managers

An application can make a call on an object reference to **`get_domain_managers`**. This returns a list of the immediately enclosing domain managers for this object. 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 object needed to administer the policy associated with this domain. Each policy object supports the operations required to administer that policy.

Note: The policy object used for administering the policy may be the same as the one used for enforcing it, but need not be. For example, an *AccessPolicy* interface for managing the policy may be supported by a different object from the one that supports the *AccessDecision* interface used for deciding if access is allowed.

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 G, 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 administrative interfaces to set the policy.

The administrative interfaces 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 specify who can access the objects.

Administrative interfaces are defined in Section 15.6, Administrator's Interfaces, for the standard policy types, which all ORBs supporting security functionality Level 2 support.

However, 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.

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 the subsections The Model as Seen by Applications, and Administrative Model.

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

- On object invocation when invocation time policies are automatically enforced.
- 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 15-45 shows the implementation objects and services used to support secure invocations.

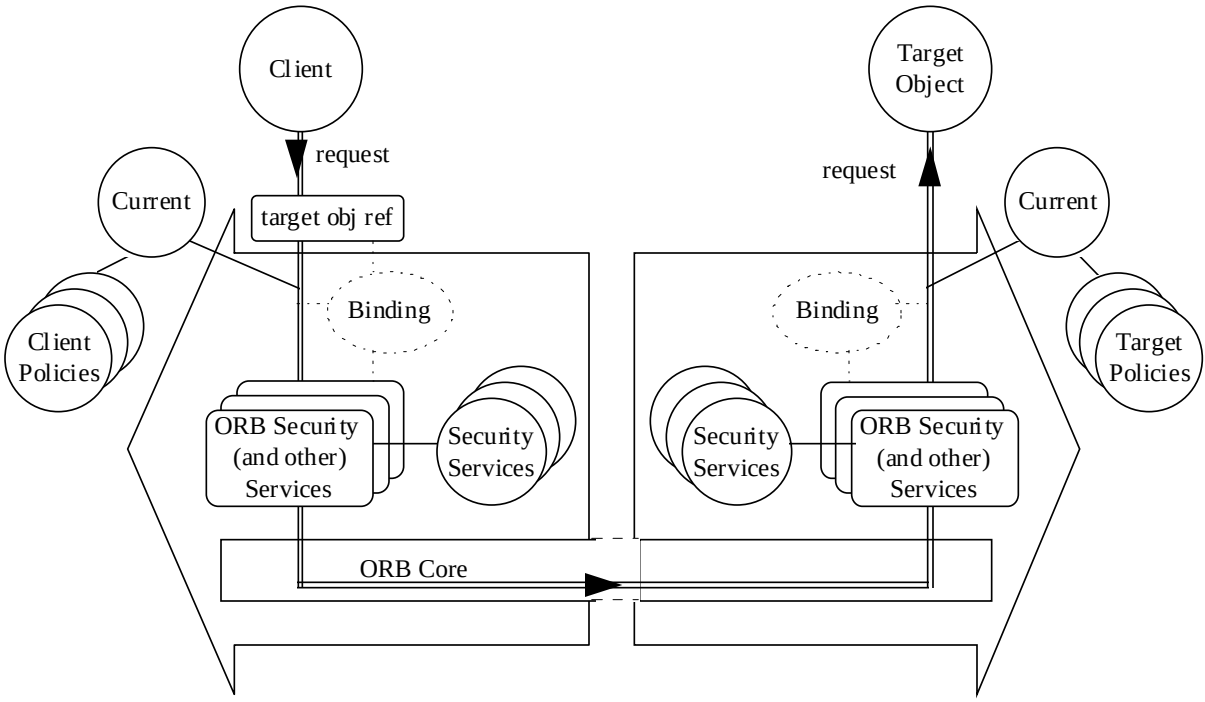

Figure 15-45 Securing invocations

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 15.4.2, Structural Model. The ORB Security Services call on the security services to provide the required security.

Security Policy

The security policies associated with the client object are accessed by the ORB Security Services using the **`get_policy`** operation on Current specifying the type of policy required. (The client side services also have to check the binding to see if any policies have been overridden by the client using operations on the target object reference.) At the target, **`get_policy`** is used on the object's reference (at least in the message level interceptors, as Current is not available at that stage).

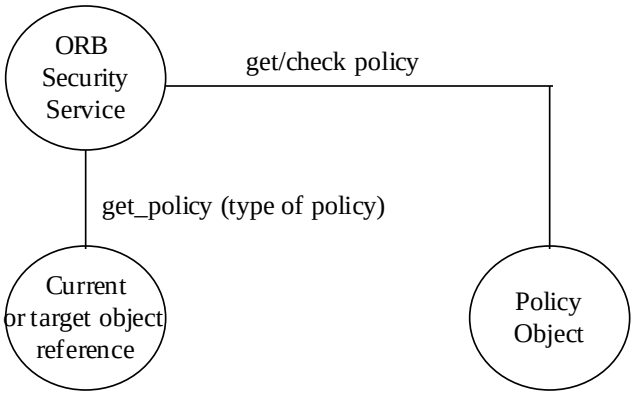

Figure 15-46 get_policy operation

The policy may be associated with domain managers as described in the administration view. However, information may be cached during environment setup or previous object invocations, and the **`get_policy`** interface hides whether the policy information has been obtained in advance or is searched for in response to this call.

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 specifying the particular policy options required (e.g. whether confidentiality is required, and the delegation mode). It then uses this information to enforce the policy, for example, pass the required policy options to the Vault to enforce.

Some policy objects may include rules, which enforce the policy. For example, an access policy object supports an **`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 15-47.

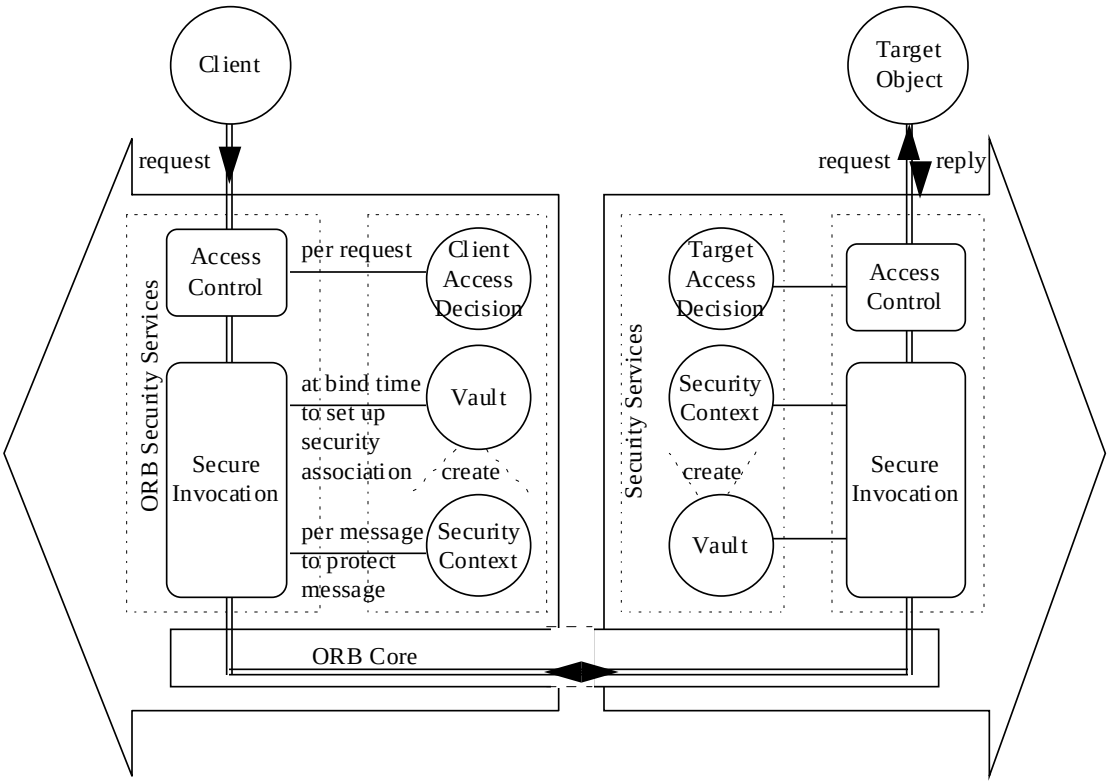

Figure 15-47 ORB Security Services

Two ORB Security Services are shown:

- The access control service, which is responsible for checking if this operation is permitted and enforcing the invocation audit policy for some event types.
- 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 this operation on this target object is permitted. It is obtained by the ORB service using the **`get_policy`** operation previously described. There may be different policies, and therefore different access decision policy objects, at the client and target.

The ORB service invokes the **access_allowed** operation on the Access Policy 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. If the Access Policy object is separate from the Access Decision object, it invokes **get_effective_rights** on the 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 G, Facilities Not in This Specification.

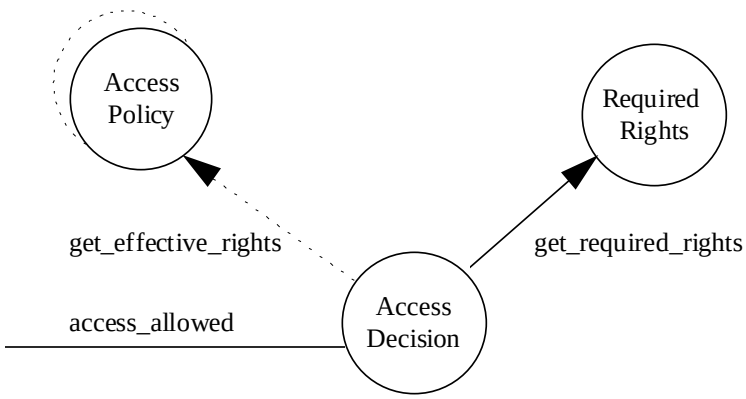

Figure 15-48 Access decision object

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 The Model as Seen by Applications, under Section 15.4.5, Security Object Models. 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. **`is_valid`** and **`refresh`** operations are supported to check the validity of the context and refresh it if possible.

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

They also have **`received_credentials`** and **`received_security_features`** attributes, which are 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 15-49. 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).

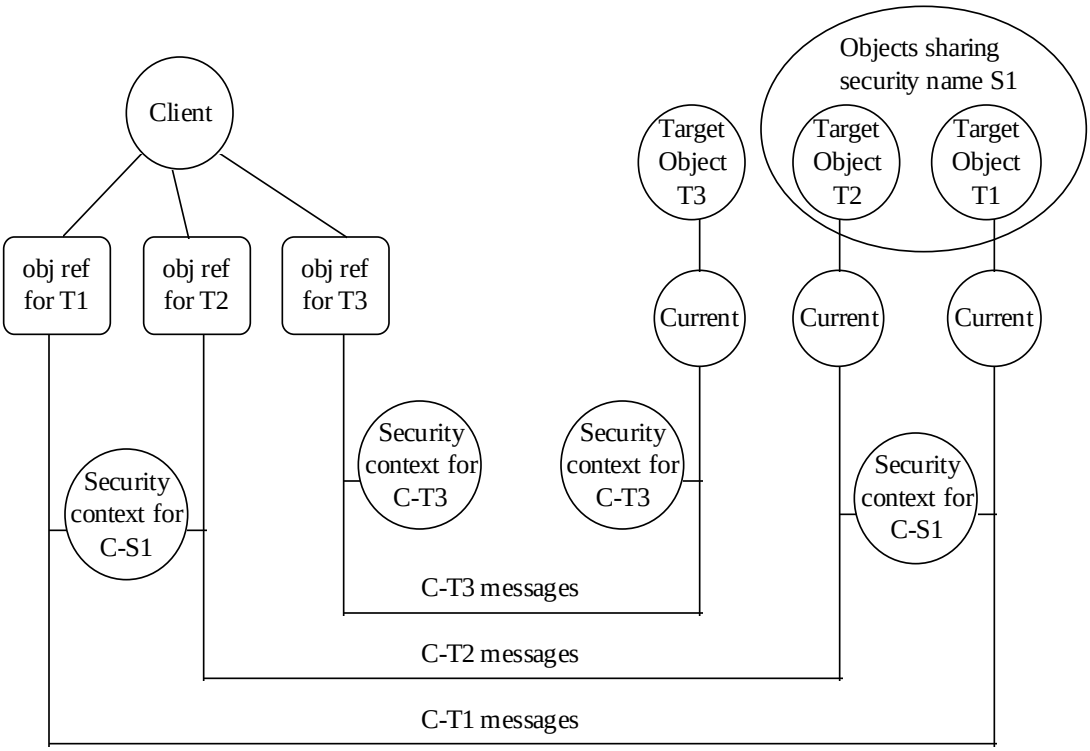

Figure 15-49 Target objects sharing security names

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 Administrative Model, in Section 15.4.5, Security Object Models.

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 15-50.

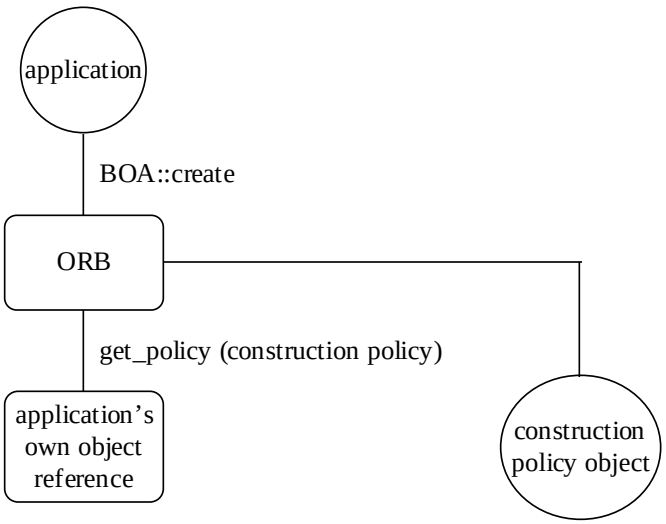

Figure 15-50 Object created by application or factory

The application (which may be a generic factory) object calls **`BOA::create`** to create the new object reference. The ORB obtains the construction policy associated with the creating object.

The construction policy controls whether, in addition to creating the specified new object, the ORB must also create a new domain for the newly created object. If a new domain is needed, the ORB creates both the requested object and a domain manager object. A reference to this domain manager can be found by calling **`get_domain_managers()`** on the newly created object's reference.

While the management interface to the construction policy object is standardized, the interface from the ORB to the policy object is assumed to be a private one, which may be optimized for different implementations.

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain.

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management interfaces.

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 15-51 shows the relationship between the main objects visible in different views for three types of security functionality.

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

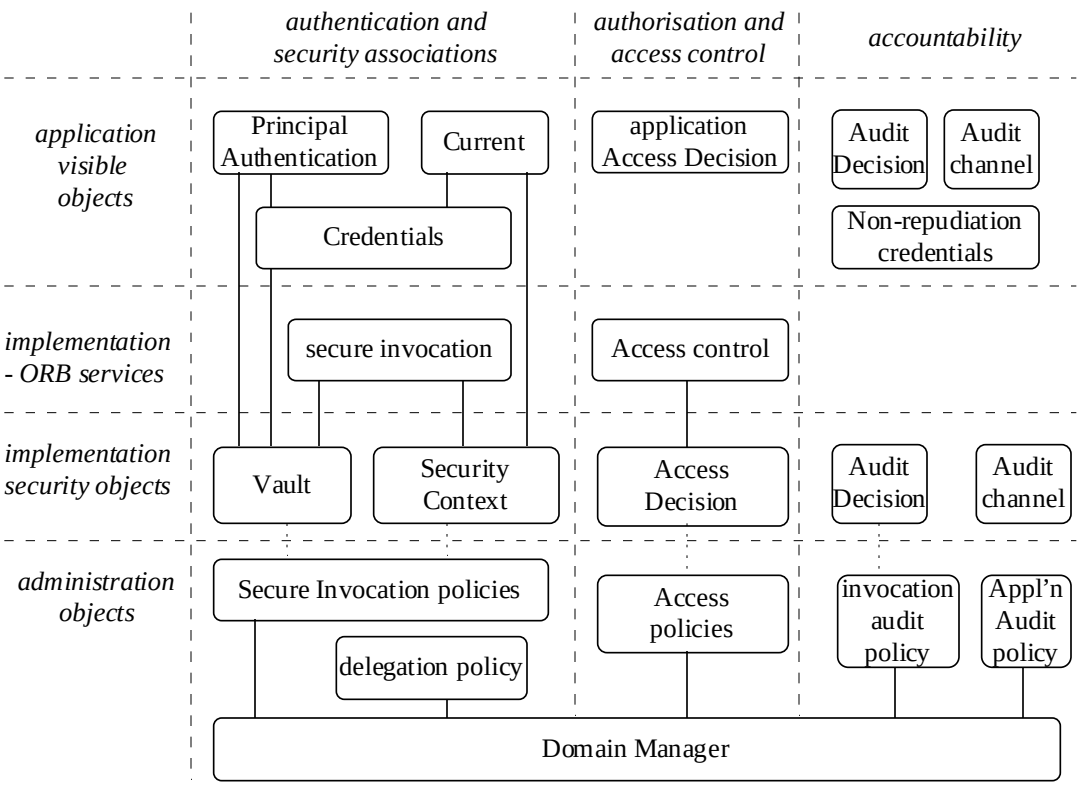

Figure 15-51 Relationship between main objects

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 interfaces to allow policies to be maintained. These interfaces 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 policy object is used, which has a standard "check if access is allowed" interface, whatever the access control policy used. The access policy object has both management and run-time interfaces.

The diagram does not show:

- Environment objects, such as Current.
- 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.

15.5 Application Developer's Interfaces

15.5.1 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 the Security Model as seen by applications in Section 15.4, Security Architecture.

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 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 **NO_IMPLEMENT** exception. No security-specific exceptions are specified.

Security Functionality Conformance

Two security functionality levels are specified in this document, plus one optional facility.

Security Functionality Level 1

Security functionality Level 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 includes interfaces for applications as follows:

- **`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 15.5.4, Interfaces, and Section 15.5.9, Use of Interfaces for Access Control).

Security Functionality Level 2

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 semantics; some implementations will therefore be capable of enforcing a wider variety of policies than others.

Security Replaceability Ready (either option)

A security replaceability ready ORB provides no security functionality itself, but is Security Ready (i.e. it makes well-formed calls to known security interfaces as defined in Appendix D, Conformance Details) and supports the **`get_service_information`** operation which allows an application to find out what security is supported (see Section 15.5.2, Finding Security Features). It also supports the **`get_current`** operation on the ORB to obtain the Current object for the execution context (see Section B.3, Extension to the Use of Current).

Optional Functionality

The only specified optional facility specified here is non-repudiation. The interfaces for this are specified in Section 15.5.11, Non-repudiation.

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.

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 *credentials* interfaces are based on those of GSS-API.

Data Types

Many of the security data types used by applications are also used for implementation interfaces. These are therefore defined in the security common data module.

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 A, Consolidated OMG IDL.

```
module CORBA {
// the following data structures are used to return what
// security is implemented by get_service_information

typedef unsigned short ServiceType;

const ServiceType Security = 1;
// other Service types to be defined

typedef unsigned long ServiceOption;

const ServiceOption         SecurityLevel1 = 1;
const ServiceOption         SecurityLevel2 = 2;
const ServiceOption         NonRepudiation = 3;
const ServiceOption         SecurityORBServiceReady = 4;
const ServiceOption         SecurityServiceReady = 5;
const ServiceOption         ReplaceORBServices = 6;
const ServiceOption         ReplaceSecurityServices = 7;
const ServiceOption         StandardSecureInteroperability = 8;
const ServiceOption         DCESecureInteroperability = 9;

// Service details supported by the implementation

typedef unsigned long ServiceDetailType;

// security mech type(s) supported for secure associations

const     ServiceDetailType         SecurityMechanismType = 1;

// privilege types supported in standard access policy

const     ServiceDetailType         SecurityAttribute = 2;
```

```
struct      ServiceDetail {
                ServiceDetailType       service_detail_type;
                sequence <octet>        service_detail;
            };

            struct ServiceInformation {
                sequence <ServiceOption>        service_options;
                sequence <ServiceDetail>        service_details;
            };


            // Security Policy Types supported

            enum PolicyType {
                SecClientInvocationAccess,
                SecTargetInvocationAccess,
                SecApplicationAccess,
                SecClientInvocationAudit,
                SecTargetInvocationAudit,
                SecApplicationAudit,
                SecDelegation,
                SecClientSecureInvocation,
                SecTargetSecureInvocation,
                SecNonRepudiation,
                SecConstruction
            };
};

 module Security {
            typedef string SecurityName;
            typedef sequence <octet> Opaque;

            // extensible families for standard data types

            struct ExtensibleFamily {
                unsigned short          family_definer;
                unsigned short          family;
            };

            // security association mechanism type

            typedef         string      MechanismType;
            struct          SecurityMechandName {
                MechanismType       mech_type;
                SecurityName        security_name;
            };

            typedef sequence<MechanismType>     MechanismTypeList;
            typedef sequence<SecurityMechandName>SecurityMechandNameList;

            // security attributes

            typedef unsigned long               SecurityAttributeType;
```

```
// identity 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;
```

```
        // Credential types which can be set as Current default

        enum CredentialType {
            SecInvocationCredentials,
            SecOwnCredentials,
            SecNRCredentials
        };

        // Declarations related to Rights
        struct Right {
            Extensible Family       rights_family;
            string                  right;
        };

        typedef sequence <Right> RightsList;

        enum RightsCombinator {
            SecAllRights,
            SecAnyRight
        };

        // Delegation related
        enum DelegationState {
            SecInitiator,
            SecDelegate
        };

// 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
        };

        // Security feature-value
        struct SecurityFeatureValue {
            SecurityFeature             feature;
            boolean                     value;
        };

        typedef sequence<SecurityFeatureValue>SecurityFeatureValueList;
```

```
// Quality of protection which can be specified
// for an object reference and used to protect messages
enum QOP {
        SecQOPNoProtection,
        SecQOPIntegrity,
        SecQOPConfidentiality,
        SecQOPIntegrityAndConfidentiality
};

// 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;

// 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
};

// 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;

        // Audit data structures

        struct AuditEventType {
            ExtensibleFamily            event_family;
            unsigned short              event_type;
        };

        typedef sequence<AuditEventType>AuditEventTypeList;

        typedef unsigned long           SelectorType;

        const   SelectorType    InterfaceRef = 1;
        const   SelectorType    ObjectRef = 2;
        const   SelectorType    Operation = 3;
        const   SelectorType    Initiator = 4;
        const   SelectorType    SuccessFailure = 5;
        const   SelectorType    Time = 6;

        // values defined for audit_needed and audit_write are:
        // InterfaceRef: object reference
        // 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

        struct  SelectorValue {
            SelectorType            selector;
            any                     value;
        };
        typedef sequence<SelectorValue> SelectorValueList;
};
```

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

15.5.2 Finding Security Features

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 **`get_service_information`** operation defined here could be used for information about other CORBA facilities and services, so is not specific to security, though only security details are specified.

Interfaces

```
interface    ORB {

    boolean get_service_information (
        in    ServiceType                    service_type,
        out   ServiceInformation             service_information,
    );
};
```

Parameters

service_type
: Identifies the service for which the information is required.

service_information
: The information pertaining to the service.

Return Value
: Returns **`TRUE`** if the service is supported and, if so, the **`service_information`** contains valid information. **`FALSE`** is returned if the service is unsupported.

Portability Implications

Applications dependent on security facilities beyond those in security functionality Level 1 may not be portable between different secure ORBs. This interface allows applications to adapt to the security available.

15.5.3 Authentication of Principals

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 15.5.11, Non-repudiation.

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.

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.

Interfaces

This section defines the “Authenticate” and “Continue Authentication” operations on the Principal Authenticator object.

authenticate

This is called, for example, by a user sponsor to authenticate the principal and optionally request privilege attributes that the principal requires during its session with the system. It creates a Credentials object including the required attributes.

```
AuthenticationStatus authenticate (
        in    AuthenticationMethod        method,
        in    string                      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.
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 Object reference of the newly created Credentials object. Not fully initialized, therefore unusable unless return parameter is 'Success.'
auth_specific_data
Information specific to the particular authentication service used.
continuation_data
If the return parameter from the authenticate operation is 'Continue,' 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 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_authentication (
    in      Opaque          response_data,
    inout   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 'Success.' Note that this parameter is described as inout, as the authentication procedure will modify the state of the Credentials object.

continuation_data
: If the return parameter from the **`continue_authentication`** operation is 'Continue,' 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). A Credentials object has therefore not been created.

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.

15.5.4 Credentials

Description of Facilities

A Credentials object represents a principal's current credential information for the session and therefore includes information such as that principal's privilege attributes and identities such as the 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.)

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.

Credential objects are created as the result of:

- Authentication (see Section 15.5.3, Authentication of Principals).
- Copying an existing Credentials object.
- Asking for a Credentials object via Current (see Section 15.5.6, Security Operations on Current).

The way these credentials are made available for use in invocations is described in Section 15.4, Security Architecture, and defined in detail in Sections 15.5.5, Object Reference, and Section 15.5.6, Security Operations on Current.

Credentials used for non-repudiation also support further facilities as described in Section 15.5.11, Non-repudiation.

Interfaces

All the following operations are part of the Credential 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 ();
```

Return Value

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

set_security_features

This associates a set of security features with a Credentials object and sets each feature to be "on" or "off." The security features affect how a secure association is set up, such as what delegation mode to use, whether trust in the target is needed, and what message protection is required.

Some implementations may allow the security features to be set for communication in one direction only (i.e. for requests only, or replies only) via the direction parameter, but this support is not required for compliant implementations. The request-only and reply-only feature sets are treated as overrides to the "both" feature set. If an unsupported direction is passed to **`set_security_features`**, the **`BAD_PARAM`** exception should be raised.

The value of a security feature set by this operation is used for invocations using this Credentials object (if this does not contravene the ClientSecureInvocation policy for that feature or the target's invocation policy). Once associated with the Credentials object, a feature may be turned "on" or "off" again with an additional call to **`set_security_feature`**.

```
void set_security_features (
        in    CommunicationDirection          direction,
        in    SecurityFeatureValueList        security_features
);
```

Parameters

direction — The communication direction (i.e. both, request, or reply) to which the security feature should be applied. Normally set to both.

security_features — A sequence of required feature-value pairs. They may indicate the delegation mode or a secure association option such as a message protection requirement, or whether trust in the target is needed. To set the feature on, set the boolean value to **`TRUE`**; a value of **`FALSE`** is used to turn off the feature.

get_security_features

This returns the security features associated with the Credentials.

The 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_features`**, the **`CORBA::BAD_PARAM`** exception should be raised.

```
SecurityFeatureValueList get_security_features (
        in    CommunicationDirection                direction
);
```

Parameters

direction — The communication direction (i.e. both, request, or reply) for which the security features should be retrieved. Normally set to both.

Return Value

A sequence of required feature-value pairs. A boolean value of **TRUE** indicates the feature is on; a value of **FALSE** indicates the feature is off.

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.)

Note: This operation is also available on the Current pseudo-object.

```
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.

refresh

This allows the application to update expired Credentials.

```
boolean refresh ();
```

Return Value

False	The Credentials could not be refreshed.

Portability Implications

The **`authenticate`** and **`set privilege`** 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).

15.5.5 Object Reference

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. The following operations are available.

- **override_default_credentials** specifies a Credentials object to be used when calling this target object. For example, the client may want to make different privileges available to different targets, so choose Credentials with the required privileges.
- **override_default_QOP** specifies that a particular quality of protection is required for future messages it sends using this object reference.
- **get_active_credentials** returns the active credentials to be used for invocations via this target object reference.
- **get_security_features** returns the quality of protection and other security features which will apply to invocations via this object reference.
- **get_policy** is used to find the security policy of the specified type for this object.

- **get_security_mechanisms** returns the security association mechanisms available.
- **override_default_mechanism** allows a different mechanism to be requested.
- **get_security_names** returns the security name(s) for the target.

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.

Although these operations quote 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.*

A target object can influence the security policy for incoming invocations by setting security policies using the administrative interfaces in Section 15.6, Administrator's Interfaces. This will affect the security information exported as part of its object reference.

Interfaces

In ORBs providing security, the Object interfaces includes the security-related interfaces defined in this section. The availability and functionality of specific operations will vary depending on the level of security provided by the ORB. OMG IDL values for defined security levels are described in Appendix A, Consolidated OMG IDL.

override_default_credentials

This specifies a Credentials object to be used for future invocations that this client makes on this target object. The client can choose any Credentials object available to it. For example, it may want to enforce a least privilege policy, so use Credentials with only those privileges required by that target object.

If needed, **`override_default_credentials`** should be used before making any invocation on this object, as it will generally result in a new security association needing to be established with the target object.

```
void override_default_credentials (
        in Credentials creds
);
```

Parameters

creds	The object reference of the Credentials object, which is to become the default.

override_default_QOP

The client application requests the quality of protection to use for messages when invoking the target object, consistent with its controlling security policy. Note that a request for a particular quality of protection may be overridden by Security Policy. For example, Security Policy may insist that all messages be confidential even if the client had not asked for this. (The invoker can determine this by calling **`get_security_features`** and reading the value actually set for it.)

It is possible to use this operation to change the QOP (e.g. confidentiality), for a particular message or sequence of messages, and then call **override_default_QOP** again to revert to a different set of options. Changing QOP will not in general require the establishment of a different security association.

This operation does not allow QOP to be overridden for a single direction of communication (i.e. request or reply). If that feature is required, use **set_security_features** on an override Credentials object.

```
void override_default_QOP (
    in      QOP             qop
);
```

Parameters

qop Required quality of protection of messages.

get_security_features

This is used by the client to find its net security requirements for invoking a particular target object, as successfully requested thus far. Note that although the operation quotes the target object reference, the scope of the effect of the operation is the use of that reference itself, and not the object it represents.

The 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_features**, the **CORBA::BAD_PARAM** exception should be raised.

```
SecurityFeatureValueList get_security_features (
    in  CommunicationDirection      direction
);
```

Parameters

direction The communication direction (i.e., both, request, or reply) for which the security features should be retrieved. Normally set to both.

Return Value

The sequence of feature-value pairs currently requested on this object reference. A boolean value of **TRUE** indicates the feature is on; otherwise **FALSE**.

get_active_credentials

This operation returns a reference to the credentials that will be used when invoking operations using this object reference.

```
Credentials get_active_credentials ();
```

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.

`get_policy` 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 that can be used to obtain the policy object.

get_security_mechanisms

Applications do not normally need to be aware of the security mechanisms used for security of the invocation between client and target. The client environment knows what mechanisms it supports, and the target object reference exported from a secure system specifies what mechanisms the target supports. So the client's ORB can normally choose the mechanism to use. Even if it cannot, negotiation of mechanisms may be supported without the application seeing it.

Applications can call **`get_security_mechanisms()`** to determine the set of mechanisms supported by both the client and the target.

```
MechanismTypeList       get_security_mechanisms();
```

Return Value

The mechanism types that both the client and target object support.

override_default_mechanism

For the rare cases where the application wants to influence what security mechanism will be used for future invocations, the application can ask to override the mechanism chosen by the system. This will apply only to invocations that this client makes using this object reference.

```
void override_default_mechanism (
    in  MechanismType       mechanism_type );
```

get_security_names

This operation is for use by security sophisticated applications. It is used by clients who wish to determine which security names are associated with the target. It is possible for different security names to be used for the target, depending on the mechanism used for the target. The name may be shared by several objects.

```
SecurityMechandNameList get_security_names ();
```

Return Value

A list of pairs of values, each containing a security mechanism and associated security name.

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.

15.5.6 Security Operations on Current

Description

The Current object represents service specific state information associated with the current execution context; both clients and targets have Current objects representing their execution contexts. (Note that a reference to the Current object representing the active execution context can be retrieved using the **`ORB::get_current()`** operation; see Section B.3, Extension to the Use of Current, for details). In a secure ORB, the Current object includes operations relevant to Security; these operations are described in this section and provide access to information about one or more of the following credentials.

- **invocation credentials**: these are the credentials at the client, used when this client invokes another object. There must always be credentials available for invocations, but setting these is generally done transparently to the business applications. When a user logs on, the user sponsor or other logon program normally sets this to the user's credentials. If this is done outside the object system, it is picked up at ORB initialization. At an intermediate object, its default value is either the received credentials or the object's own credentials, depending on the delegation policy that applies to that object.
- **own credentials**: the credentials associated with the active object. A particular object may have its own credentials or may share credentials with other objects. An object's own credentials are normally set up as the result of the object (or the environment domain to which it belongs) being initialized.

- **received credentials**: the credentials received from the client of the invocation as seen at the target object. Depending on delegation options, this may be a single Credentials object, or a list of credentials including those of both the initiator and other principals in the chain
- **non-repudiation credentials:** when non-repudiation is supported**,** the credentials of the initiating principal in whose name evidence is being generated or verified. On logon, or ORB initialization, these are normally set to the user's credentials. At other objects, they are set by default to their own credentials.

The following applications have the following functions.

- **get_attributes** obtain privilege and other attributes associated with received credentials (which should be the user's privileges when at the workstation).
- **set_credentials** 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.
- **get_credentials** can obtain the credentials currently associated with the Current object for invocation, non-repudiation, or as its own credentials.
- **received_credentials** attribute contains the credentials received from the client.
- **received_security_features**, an attribute at the target application, contains the security features of the message sent by the client.

The application can also use the

- **get_policy** operation to find what security policies apply to it.
- **required_rights_object** attribute to discover which operations require which rights.
- **principal_authenticator** attribute to get a reference to a PrincipalAuthenticator object (which can be used to authenticate principals and thus obtain Credentials objects for them).

Interfaces

get_attributes

This 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 interface 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.

Note that a **get_attributes** operation is also available on Credential objects.

```
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.

set_credentials

As described previously, 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. **set_credentials** on the Current object sets the specified credentials as the default one for the following.

- Subsequent *invocation*s made by that client.
  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 **received_credentials** as the specified credentials on **set_credentials**.
- The object's *own* credentials.
  If an application authenticates itself (so creates new credentials), or sets privileges on its own credentials, getting a new credentials object, it can use **set_credentials** to set these credentials as its own on invocations requiring them (e.g. for composite delegation).
- Non-repudiation.
  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.

```
void set_credentials (
        in     CredentialType          cred_type,
        in Credentials                 creds
);
```

Parameters

cred_type — The type of credential to be set (i.e. invocation, own, or non-repudiation).

creds — The object reference of the Credentials object, which is to become the default.

get_credentials

This operation allows an application access to the credentials associated with Current. As for **`set_credentials`**, the application can ask for the default credentials for future invocations, its own credentials, 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).

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

Parameters

cred_type The type of credentials to be obtained.

Return Value

The object reference of the credentials.

received_credentials

At a target object, this gets the credentials received from the client. If credentials representing more than one principal are received, the contents of these credentials depend on the delegation model in use. They may be:

- The credentials of the only principal identified, if simple delegation is used (or if the security technology used has merged the credentials of all the callers in the chain).
- A list of credentials, if the credentials for different principals in a chain of calls can be distinguished. Note that the number of credentials in this list depend on the delegation option in use. There may be credentials for the initiator of the chain and the immediate invoker only, or credentials providing a trace of all principals in the chain. The first entry in the chain is the "primary" principal's credentials, normally the credentials of the initiator of the chain. A **`get_attributes`** call on Current returns the privileges from these credentials.

At the workstation, the **`received_credentials`** attribute is the user's credentials, which are also the default credentials for invocation.

```
readonly attribute CredentialsList received_credentials;
```

Return Value

A sequence of Credential object references received from the requester.

received_security_features

This attribute at the target application provides the security features of the message sent by the client.

```
readonly attribute SecurityFeatureValueList
    received_security_features ;
```

Return Value

A sequence of feature-value pairs. A boolean value of **TRUE** indicates that the feature is on; otherwise **FALSE**.

get_policy

This gets the security policy object of the specified type, which applies to this object. When used on Current, it gets the security policy object for this client (which may not be an object) or the current object.

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

Parameters

policy_type The type of policy to be obtained.

Return Value

policy A policy object which can be used to interrogate the policy in force as defined in Section 15.6, Administrator's Interfaces. 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 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, though 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;
```

The operations in the interface of this object are defined in Section 15.6.4, Access Policies.

principal_authenticator

This 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;
```

The operations in the interface of this object are defined in Section 15.3.2, Principles and Their Security Attributes.

15.5.7 Security Audit

Description of Facilities

Auditing of object invocations is done automatically by the ORB according to the audit invocation policies (ClientInvocationAudit and TargetInvocationAudit) 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 15.6.5, Audit Policies.

Interfaces

The interfaces specified here are the ones to the Audit Decision object to decide what to audit, and the Audit Channel interface used to write the audit records.

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.

```
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_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 document. It is expected to be an event service of some sort, such as an OMG 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* 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

This attribute of the Audit Decision object provides the audit channel associated with this audit policy.

```
readonly attribute AuditChannel audit_channel;
```

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.

15.5.8 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 15.6, Administrator's Interface.

15.5.9 Use of Interfaces for Access Control

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 15.3.4, Access Control Model, 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 15.6.3, Security Policies Introduction) 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 15.6, Administrator's Interface). 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.

Interfaces

Application access decision functions should be made by 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 should support an **`access_allowed`** operation as is used for enforcing access policies in the ORB (see Access Decision Object under Section 15.7.4, Implementation-Level Security Object Interfaces). 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 Interfaces under Section 15.6.2, Security Operations on Current).
  - A credentials object, which represents principal.
  - A credentials list (the **`received_credentials`**), where access controls distinguish initiator and delegate principals.
- 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 should be **TRUE** if access is permitted, otherwise **FALSE**.

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.

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.

15.5.10 Use of Interfaces for Delegation

Description of Facilities

An operation on a target object may result in calls on many other objects as described in Section 15.3.6, Delegation. 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.

Interfaces

All the interfaces used for delegation are specified elsewhere. This section describes how they are used during delegation.

An intermediate object can set the delegation mode for an invocation by performing a **`set_security_features`** operation on the Credentials object to be used for the invocation (see Section 15.5.4, Credentials). This can be used to set the delegation mode to:

- NoDelegation, meaning use the intermediate's object's own credentials.
- SimpleDelegation, meaning use the credentials received from the client.
- CompositeDelegation, meaning use both.

The way the received and intermediate's own credentials are combined in CompositeDelegation 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 **`received_credentials`** (see Interfaces under Section 15.5.6, Security Operations on Current), 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.

The **`received_credentials`** attribute provides a CredentialsList, not just a single Credentials object. This list will only have more than one entry after composite delegation as defined above. If there is more than one entry in the list, the first entry is that of the initiator in the chain, normally the main one used for access controls. This is also the one whose privileges are obtained via **`get_attributes`**.

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 NotSupported exception if CompositeDelegation 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 Interfaces under Section 15.5.9, Use of Interfaces for Access Control), 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.

15.5.11 Non-repudiation

Non-repudiation is an optional facility, not part of security functionality Level 1 or 2.

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.

Interfaces

Non-repudiation Service Data Types

The following data types are used in the Non-repudiation Service interfaces:

```
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
    };

// the following are used for evidence validity duration
typedef ulong 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;
    };
```

Non-repudiation Service Operations

This section describes the Non-Repudiation Service operations. All these operations are part of the interface of the NRCredentials object.

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	If the requested features were equivalent.
false	If the actual features differ from the requested features.

get_NR_features

A *get_NR_features* interface is provided to allow security-aware applications to determine what NR policy is currently in effect:

```
NRPolicyFeaturesList    get_NR_features ();
```

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 NoEvidence).

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 Valid) or an "augmented" but still incomplete evidence token, in which case conditionally valid 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 conditionallyValid, 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 conditionallyValid, 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 NoEvidence).

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**.

request_features
Describes the included request, if any. See the **`generate_NR_token`** parameter description for details.

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

input_token The 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.

15.6 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.

15.6.1 Concepts

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.

Because administrative or management services in general have been identified as a *Common Facility* in the Object Management Architecture, only minimal, security-specific interfaces are given here together.

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.

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, termed the **domain manager**, which references the policy objects for this domain, and zero or more other objects, which are domain **members** and therefore subject to the policies.

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 under Section 15.3.8, Domains, domains may be members of other domains, so 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.

Security Policies

This specification covers administration of security policies, which are enforced by a secure object system 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.

15.6.2 Domain Management

This section includes the interfaces needed to find domain managers and find the policies associated with these. However, it does not include interfaces to manage domain membership, structure of domains, and manage which policies are associated with domains, as these are expected to be developed in a future Management Facility specification (for example, one based on the X/Open Systems Management Preliminary Specification); the Collection Service is also relevant here.

This section also includes the interface to the construction policy object, as that is also relevant to domains. Similarly, it includes the interface administrative applications needed to find the domains (and therefore the policies) that apply to objects. The basic definitions of the interfaces related to these are part of the CORBA module, since other definitions in the CORBA module depend on these.

Interfaces to administer the security policy objects are defined in Section 15.6.3, Security Policies Introduction.

```
module CORBA                // Basic Management infrastructure
{
    interface Policy                // Features common to all Policies
    {    };

    interface DomainManager {
    // Features common to all Domain Managers
```

```
        // get policies for objects in this domain; each domain may have
        // policies of various different types. This call returns the policy
        // of the specified type for the domain which is the target of the call.
            Policy      get_domain_policy (
            in PolicyType                   policy_type);
        // Note that the domain manager also inherits the
        // get_policy and get_domain_managers operations
        // defined for all objects in a secure system - see below
        };

        interface ConstructionPolicy: Policy{
            void        make_domain_manager(
                    in CORBA::InterfaceDef    object_type);
        };

        // additions to CORBA::Object interface
        interface Object {
            DomainManagerList get_domain_managers();
        //      Note that Section 15.5 defines other extensions to
        // the Object interface, including get_policy
        };

        typedef sequence<DomainManager> DomainManagerList;
};
```

Policy

The return type of operations that retrieve policy objects. This is an empty interface from which various Policy interfaces are derived.

Domain Manager

The domain manager will provide mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

There should be no unnecessary constraints on the ordering of these activities, for example, it must be possible to add new policies to a domain with a preexisting membership. In this case, some means of determining the members that do not conform to a policy that may be imposed is required.

All domain managers provide the **`get_domain_policy`** operation, in addition to the other policy-related operations provided by the **`CORBA::Object`** interface, i.e. **`get_policy`** (described in Section 15.5.5, Object Reference) and **`get_domain_managers`** (described in Extensions to the Object Interface under Section 15.6.2, Domain Management).

get_domain_policy

This gets the policy of the specified type for objects in this domain.

```
Policy get_domain_policy (
        in PolicyType           policy_type
);
```

Parameters

policy_type — The type of policy for objects in the domain which the application wants to administer. For security, the possible policy types are described in Section 15.6.3, Security Policies Introduction.

Return Value

A reference to the policy object for the specified type of policy in this domain.

Construction Policy

The construction policy interface allows callers to specify that when instances of a particular interface are created, they should be automatically assigned membership in a newly created domain at creation time.

make_domain_manager

This specifies that when an instance of the interface specified by the input parameter is created, a new domain manager will be created and the newly created object will respond to **`get_domain_managers`**() by returning a reference to this domain manager. This policy is implemented by the ORB during execution of **`BOA::create`** (or equivalent) and results in the construction of both the application-specified object and a Domain Manager object.

```
void make_domain_manager (
        in      InterfaceDef            object_type
);
```

Parameters

object_type — The type of the objects for which Domain Managers will be created. If this is nil, the policy applies to all objects in the domain.

Extensions to the Object Interface

Section 15.5.5, Object Reference, defines operations on the **`CORBA::Object`** interface for application use. Note that these include a **`get_policy`** operation. For administrative applications, the Object interface is also extended with the following operation.

get_domain_managers

`get_domain_managers` allows security administration services (and security-aware applications) to retrieve the domain managers, and hence the security policies applicable to individual objects.

```
sequence <DomainManager> get_domain_managers ();
```

Return Value

The list of immediately enclosing domain managers of this domain manager.

15.6.3 Security Policies Introduction

Invocation security policies are enforced automatically by ORB services during object invocation. These are:

- **invocation access** policies (ClientInvocationAccess and TargetInvocationAccess) for controlling access to objects.
- **invocation audit** policies (ClientInvocationAudit and TargetInvocationAudit) control which operations on which objects are to be audited.
- **invocation delegation** policies for controlling the delegation of privileges.
- **secure invocation** policies (ClientSecureInvocation and TargetSecureInvocation) 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, which applications may use to manage and enforce their access policies.
- **application audit** policy, which applications can use to manage and enforce their audit policies.
- **non-repudiation** policies determine the rules for the generation and use of evidence.

There is also a policy concerned with creation of objects, which is enforced by **`BOA::create`**. This is the **construction policy**, which controls whether a new domain is created when an object of a specified type is created.

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.

Interfaces are provided for setting all the types of security policies previously listed. In each case, these management interfaces 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 whose interfaces are defined in this specification with different policy objects supporting different semantics; in general such policy objects will also have management interfaces different from those defined in this specification.

15.6.4 Access Policies

There are two invocation access policies: the ClientInvocationAccess policy, which is used at the client side of an invocation, and the TargetInvocationAccess policy, 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 15.5, Application Developer's Interfaces); rights are described in the next section.

Rights

The standard AccessPolicy objects in a secure CORBA system implement access policy using *rights* (though implementations may define alternative, non-rights-based AccessPolicy objects).

In rights-based systems, AccessPolicy 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 RequiredRights object is available as an attribute of Current in every execution context. Every RequiredRights 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 in DomainAccessPolicy Interface 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 three rights: “g” (interpreted to mean “get”), “s” (interpreted to mean “set”), and “m” (interpreted to mean “manage”). Implementations may define additional Rights Families. Rights are always qualified by the RightsFamily to which they belong.

RequiredRights Interface

A RequiredRights object can be thought of as a table; an example RequiredRights table appears later in this section. Note that implementations need not manage required rights on an interface-by-interface basis; RequiredRights 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 RequiredRights object instance, or by one of a number of replicated instances of a master RequiredRights object instance.

An operation’s entry in the RequiredRights 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 *operationName* 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 Identifier           operation_name,
    in 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.
rights	The returned list of required rights.
rights_combinator	The returned rights combinator.

set_required_rights

This operation updates the rights required to execute the operation specified by *operationName* of the interface specified by *interface*. 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 RequiredRights 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 string           operation_name,
    in 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 **`rights_combinator`**.

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 credential by an invocation access policy. It inherits the Policy interface and has one operation, **`get_effective_rights`**.

get_effective_rights

This operation returns the current effective rights (of family *RightsFamily*) granted by this AccessPolicy object to the subject possessing all privilege attributes in the credentials *cred*.

```
RightsList get_effective_rights (
in CredentialsList                creds_list,
in ExtensibleFamily               rights_family
);
```

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 Credential (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.

Specific Invocation Access Policies

This specification allows different Invocation Access policies to be provided through specialization of the AccessPolicy interface.

Each specific Invocation Access policy is responsible for defining its own administrative interfaces. The specification defines a standard Invocation Access policy interface, including administrative operations; it is presented in the next section. This standard policy may of course be replaced by or augmented with other policies.

DomainAccessPolicy Interface

The *DomainAccessPolicy* interface provides discretionary access policy management semantics. CORBA implementations with policy requirements, which cannot be met by the DomainAccessPolicy 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 DomainAccessPolicy 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) to perform operations on the “objects” in the domain. A DomainAccessPolicy 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 15-1 (note that the use of the Delegation State will be discussed in the section of the same name next).

Table 15-1 DomainAccessPolicy

Subject	Delegation State	Granted Rights
alice	initiator	corba:gs-
bob	initiator	corba:g--
cathy	initiator	corba:g--
...		
zeke	initiator	corba:gs-

This DomainAccessPolicy 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 DomainAccessPolicy’s standard “corba” Rights Family. The delegation state column is described under the heading “Delegation States”.)

DomainAccessPolicy Use of Privilege Attributes

Administration of principals by individual identity is costly, so the DomainAccess 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 Access Policies under Section 15.3.4, Access Control Module). 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 DomainAccessPolicy 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 15-2 User Privilege Attributes (Not Defined by This Specification)

Users	Privilege Attribute
bob, cathy	group:programmers
zeke	group:administrators

Given the definitions in this table, we can simplify our DomainAccessPolicy as follows (note that, for convenience, each PrivilegeAttribute entry is annotated in the table with its PrivilegeAttribute type).

Table 15-3 DomainAccessPolicy (with Privilege Attributes)

Privilege Attribute	Delegation State	Granted Rights
access_id:alice	intitator	corba:gs-
group:programmers	initiator	corba:g--
group:administrators	initiator	corba:gs-

Delegation State

The DomainAccessPolicy 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 DomainAccessPolicy shown next illustrates the use of this feature.

Table 15-4 DomainAccessPolicy (with Delegate entry)

Privilege Attribute	Delegation State	Granted Rights
access_id:alice	initiator	corba:gs-
access_id:alice	delegate	corba:g--
group:programmers	initiator	corba:g--
group:administrators	initiator	corba:gs-

This DomainAccessPolicy 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.

DomainAccessPolicy Use of Rights and Rights Families

The rights granted to a Privilege Attribute by a DomainAccessPolicy entry must each be “tagged” with the RightsFamily to which they belong; each DomainAccessPolicy 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.

AccessDecision Use of AccessPolicy and RequiredRights

The AccessDecision object and its interfaces are described in Access Decision Object under Section 15.7.4, Implementation-Level Security Object Interfaces. It is used at run-time to perform access control checks. AccessDecision objects rely upon AccessPolicy

objects to provide the policy information upon which their decisions are based (some implementations may provide both the AccessDecision and AccessPolicy interfaces on the same object).

To complete the example, imagine that we have the following set of object instances.

Table 15-5 Interface Instances

Objects	**Interface**
obj_1, obj_8, obj_n	c1
obj_2, obj_5	c2
obj_12	c3

The DomainAccessPolicy object illustrated next has been updated to include a list of rights of type “other” granted to each of the Privilege attributes.

Table 15-6 DomainAccessPolicy (with Required Rights Mapping)

Privilege Attribute	**Delegation State**	**Granted Rights**
access_id:alice	initiator	corba: gs- other: -u-m-s
access_id:alice	delegate	corba: g-- other: ------
group:programmers	initiator	corba: g-- other: -u----
group:administrators	initiator	corba: gs- other: ------

Table 15-7 shows RequiredRights () for three object interfaces (c1, c2, and c3), using the standard RightsFamily “corba” and a second RightsFamily, “other,” whose rights set is assumed to be {g, u, o, m, t, s}.

Table 15-7 RequiredRights 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 DomainAccessPolicy.

- 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.

DomainAccessPolicy Interface

The DomainAccessPolicy object provides interfaces for managing access policy.

Each domain manager may have at most one AccessPolicy, and therefore at most one DomainAccessPolicy (though an object instance may have more than one domain manager, and therefore, more than one DomainAccessPolicy). 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  ExtensibleFamily       rights_family,
        in  RightsList             rights
    );
```

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 ExtensibleFamily        rights_family,
    in RightsList              rights
    );
```

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 ExtensibleFamily        rights_family,
in RightsList              rights
);
```

get_rights

This operation returns the current rights (of type *rightsFamily*) 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
    );
```

15.6.5 Audit Policies

There are two invocation audit policies: the ClientInvocationAudit policy, which is used at the client side of an invocation, and the TargetInvocationAudit 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.

Audit Administration Interfaces

A standard invocation audit policy administration interface is part of Security Functionality Level 2. It 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 on **`audit_needed`** (on AuditDecision objects) and **`audit_write`** (on Audit Channel objects) as follows.

Table 15-8 Standard Audit Policy

Selector Type	Value on audit_needed and audit_write	Value Administered
Interface	from object reference	object type
Object	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

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 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::InterfaceDef      object_type,
        in   AuditEventTypeList       events,
        in   SelectorValueList        selectors
);
```

Parameters

object_type	The type of objects for which an audit policy is being set. If this is nil, all object types are implied.
events	Event types are specified as family and type ids. If the type id is zero, the selectors apply to all event types in that family.
selectors	The values of the selectors to be used.

clear_audit_selectors

This clears all audit selectors for the specified event families and types.

```
void clear_audit_selectors (
        in    CORBA::InterfaceDef          object_type,
        in    AuditEventTypeList           events,
);
```

replace_audit_selectors

This replaces the specified selectors.

```
void replace_audit_selectors (
        in    CORBA::InterfaceDef          object_type,
        in    AuditEventTypeList           events,
        in    SelectorValueList            selectors
);
```

get_audit_selectors

This obtains the current values of the selectors for the specified event family or event.

```
SelectorValueList get_audit_selectors (
        in    CORBA::InterfaceDef          object_type,
        in    AuditEventTypeList           events,
        in    SelectorValueList            selectors
);
```

set_audit_channel

This specifies the audit channel object to be used with this audit policy.

```
void set_audit_channel (
                in    AuditChannel         audit_channel
);
```

15.6.6 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:

- ClientSecureInvocation policy, which specifies the client policy in terms of trust in the target's identity and protection requirements of the communications between them.
- TargetSecureInvocation policy, which specifies the target policy in terms of trust in the client's identity and protection requirements of the communications between them
- Delegation 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 interfaces allow support of a range of policies.

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 ClientSecureInvocation and TargetSecureInvocation 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 ClientSecureInvocation and TargetSecureInvocation 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.

Secure Association Options

The selectable secure association options 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.

Secure Invocation Administration Interfaces

Set Association Options

This method on the ClientSecureInvocation and TargetSecureInvocation policy objects 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 TargetSecureInvocation policy, is not meaningful as a requirement for the ClientSecureInvocation 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::InterfaceDef          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.

get_association_options

This is used to find what secure association options apply on ClientSecureInvocation and TargetSecureInvocation 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::InterfaceDef          object_type,
        in RequiresSupports             requires_supports,
        in InvocationDirection          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.

Invocation Delegation Policy

This policy 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::InterfaceDef       object_type,
        in    DelegationMode            mode
);
```

Parameters

object_type — The type of the objects to which this delegation policy applies.

mode — The delegation mode. Options are:
SecDelModeNodelegation: the intermediates's own credentials are used for future invocations.
SecDelModeSimple: the initiating principal credentials are delegated.
SecDelModeComposite: 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.

get_delegation_mode

This returns the delegation mode associated with the object.

```
DelegationMode get_delegation_mode (
        in      CORBA::InterfaceDef             object_type
);
```

15.6.7 Non-repudiation Policy Management

This section defines interfaces for management of non-repudiation policy. These interfaces are included in the non-repudiation conformance option.

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.

Data Types for Non-repudiation Policy Management Interfaces

The following data types are used by the NR policy management interfaces.

```
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;
```

Non-repudiation Policy Management Interfaces

The non-repudiation policy defined in this specification supports **`get_NR_policy_info`** and **`set_NR_policy_info`** operations.

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.

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 Values

true If the requested mechanisms were all set.
false If the actual mechanisms returned differ from those requested.

15.7 Implementor's Security Interfaces

This section describes the ORB facilities available to security service implementors to support construction of secure ORBs using portable components and also the object security services, which implement security. The interfaces defined in this appendix support the replaceability conformance options defined in Appendix D, Conformance Details.

- Generic ORB service (interceptor) interfaces. This section defines ORB interfaces that allow services such as security to be inserted in the invocation path. Interceptors are not specific to security; they could be used to invoke any ORB service. Interceptors are therefore proposed as a generic ORB extension. For this reason, the generic interfaces supported by interceptors are presented in Appendix B, Summary of CORBA 2 Core Changes; only security-specific interceptor

interfaces are defined in this section. These interfaces permit services to be neatly separated so that, for example, security functions can coexist with other ORB services such as transactions and replication (see Section 15.7.1, Generic ORB Services and Interceptors).

- Security Service replaceability. This appendix 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 15.7.4, Implementation-Level Security Object Interfaces, 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 15.7.3, Security Interceptors (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.

None of the interfaces defined in this section affect the application and administrator's views described in Section 15.5, Application Developer's Interfaces, and Section 15.6, Administrator's Interfaces.

15.7.1 Generic ORB Services and Interceptors

An Interceptor implements one or more ORB services. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and target object. Two types of interceptors are defined in this specification:

- **Request-level interceptor**, which perform transformations on a structured request.
- **Message-level interceptors**, which perform transformations on an unstructured buffer.

Figure 15-52 shows interceptors being called during the path of an invocation.

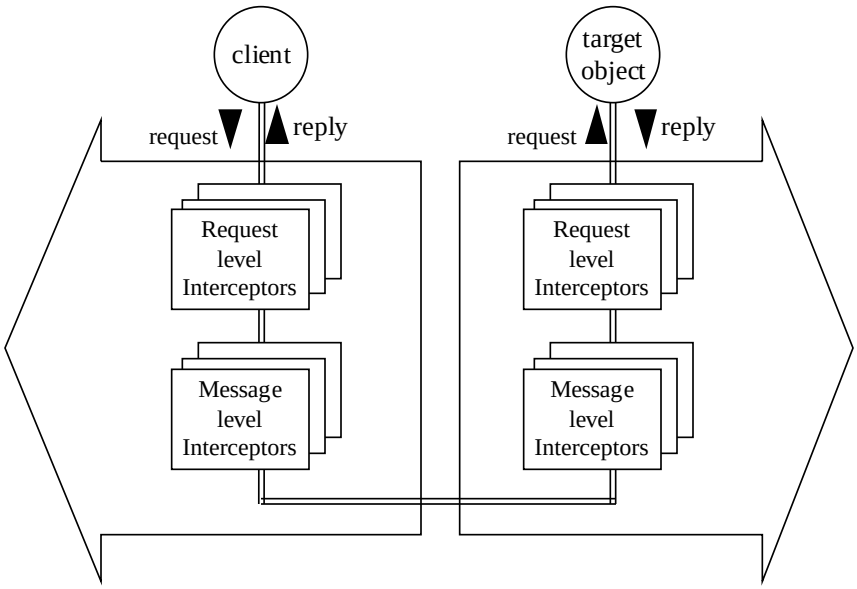

Figure 15-52 Interceptors Called During Invocation Path

15.7.2 Request-Level Interceptors

Request-level interceptors are used to implement services which may be required regardless of whether the client and target are collocated or remote. They resemble the CORBA bridge mechanism in that they receive the request as a parameter, and subsequently reinvoke it using the Dynamic Invocation Interface (DII). An example of a request-level interceptor is the Access Control interceptor, which uses information about the requesting principal and the operation in order to make an access control decision.

The ORB core invokes each request level interceptor via the **`client_invoke`** operation (at the client) or the **`target_invoke`** operation (at the target) defined in this section. The interceptor may then perform actions, including invoking other objects, before reinvoking the (transformed) request using **`CORBA::Request::invoke`**. When the latter invocation completes, the interceptor has the opportunity to perform other actions, including recovering from errors and retrying the invocation or auditing the result if necessary, before returning.

Message-Level Interceptors

When remote invocation is required, the ORB will transform the request into a message, which can be sent over the network. As functions such as encryption are performed on messages, a second kind on interceptor interface is required.

The ORB code invokes each message-level interceptor via the **`send_message`** operation (when sending a message, for example, the request at the client and the reply at the target) or the **`receive_message`** operation (when receiving a message). Both have a message as an argument. The interceptor generally transforms the message and then

invokes send. Send operations return control to the caller without waiting for the operation to finish. Having completed the **send_message** operation, the interceptor can continue with its function or return.

Selecting Interceptors

An ORB that uses interceptors must know which interceptors may need to be called, and in what order they need to be called. An ORB that supports interceptors, when serving as a client, uses information in the target object reference, as well as local policy, to decide which interceptors must actually be called during the processing of a particular request sent to a particular target object.

When an interceptor is first invoked, a bind time function is used to set up interceptor binding information for future use.

Interceptor Interfaces

This section describes the interceptors defined specifically for invoking the security services.

Details of the interfaces common to all interceptors are included in Appendix B, Summary of CORBA 2 Core Changes, as they are not security-specific. Appendix B includes definitions of:

- The RequestInterceptor interfaces **client_invoke** and **target_invoke**.
- The MessageInterceptor interfaces, including **send_message** and **receive_message**.

Appendix B also describes which interfaces the interceptors call, e.g. to get information from the tags in an IOR. Some extensions are proposed to these CORBA interfaces to give access to other information not currently in the CORBA 2 specification, such as the component tags of a multicomponent profile in an object reference.

15.7.3 Security Interceptors

The ORB Services replaceability option requires implementation of two security interceptors:

- **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 15-53.

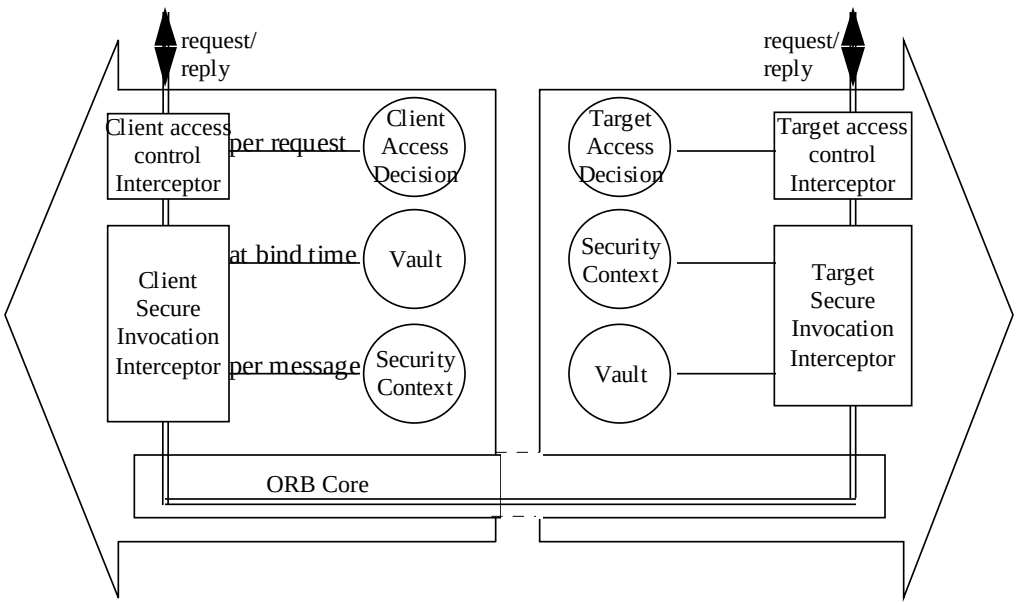

Figure 15-53 Security Functionality Implemented by Security Service Objects

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.

Invocation Time Policies

Interceptors decide what security policies to enforce on an invocation as follows:

- They call the **`get_policy`** operation defined in Section 15.5, Application Developer's Interfaces, 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 **`get_credentials`** on Current and **`get_security_features`** on the object reference.

The **`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 **`get_association_options`** as defined in Section 15.6, 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 **`get_delegation_mode`** to retrieve this information.

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.

Note: 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.

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, such as those caused by the client invoking **`override_default_credentials`**.

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 Establishing Security Associations under Section 15.3.3, Secure Object Invocations).

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:

- 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).
- 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 15.8, Security and Interoperability.

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 quality of protection required for the request may have changed since the last invocation in this security association, as the client may have used **`override_default_QOP`** to set the QOP on the target object reference. The interceptor therefore has to get the QOP by using **`get_security_features`** on the object reference. The interceptor should also check if **`override_default_credentials`** has been used, and if so, set up a new security association as at bind time.

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.

Access Control Interceptor

Bind Time

At bind time, the client access control interceptor uses **`Current::get_policy`** to get the ClientInvocationAccess Policy and ClientInvocationAudit policy. The target access

control interceptor uses the **get_policy** interface on the target object reference to get the TargetInvocationAccessPolicy and TargetInvocationAudit 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 Policy object obtained at bind time, specifying the appropriate authorization data. The access policy 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 Policy object supports. It may be ACL-based, capability-based, label-based, etc. It also does not know if the Access Policy 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 appropriate Audit Policy object; if this call indicates that the invocation attempt should be audited, the Access Control Interceptor calls the *Audit Channel* interface to write the appropriate audit record.

This interceptor does not transform the request. It either passes the request unchanged when using **CORBA::Request::invoke** to continue processing the request, or it aborts the request by returning with an exception, rather than calling **CORBA::Request::invoke**.

15.7.4 Implementation-Level Security Object Interfaces

This specification defines four 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.
- *Access Decision* objects are used (usually by Access Control Interceptors) to decide if requests should be allowed or disallowed.
- *Audit.* Audit Decision objects are used to decide if events are to be audited, and Audit Channel objects are used to write audit records to the audit trail.

Vault

The *Vault* interface provides methods for establishing security contexts between clients and targets when these are in different trust domains, so that authentication is required to establish trust. Implementations of the *Vault* interface are responsible for calling **`audit_needed`** to determine whether the audit policy requires auditing of successful and/or failed access control checks, and for calling **`audit_write`** whenever audit is needed.

Interfaces

The *Vault* interfaces are described next. Note that if a call to the *Vault* interface 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.

init_security_context

This 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. As part of this, it creates the Security Context object, which will represent the client's view of the shared security context.

```
AssociationStatus init_security_context          (
    in    CredentialsList              creds_list,
    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   Opaque                       security_token,
    out   SecurityContext              security_context);
```

Parameters

creds_list — The credentials to be used to establish the security association. There is normally only one credential object: either the default ones from Current, or the ones specified in an override operation on the target object reference. However, for composite, combined or traced delegation, more than one credential object is needed.

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 15.7.3, Security Interceptors.

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
: This is the Security Context object at the client which represents the shared security context between client and target as identified by the specified security target name.

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 is used by the association interceptor (or ORB) at the target to accept a request from the client to establish a security association. As part of this, it creates the Security Context object, which will represent the target’s view of the shared security context.

```
AssociationStatus accept_security_context (
    in   CredentialsList     creds_list,
    in   Opaque              chan_bindings,
    in   Opaque              in_token,
    out  Opaque              out_token,
    out  SecurityContext     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 as at the client, 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 ();
```

Return Value

The list of mechanism types supported by this Vault object and the association options they support.

Security Context Object

A Security Context object represents the shared security context between a client and a target. 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.

Interfaces

The Security Context object has the following attributes in common with the Current object:

```
readonly attribute CredentialsList          received_credentials;

readonly attribute SecurityFeatureValueList security_features;
```

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      Opaque      in_token
    out     Opaque      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 on 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    Opaque                    message,
        in    QOP                       qop,
        out   Opaque                    text_buffer,
        out   Opaque                    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 Security Context object provides the means whereby a protected message may be checked for integrity and the message optionally decrypted if needed.

```
boolean reclaim_message (
        in    Opaque                    text_buffer,
        in    Opaque                    token,
        out   QOP                       qop,
        out   Opaque                    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 on 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

If the **is_valid** operation returns a value of **FALSE**, then the context is no longer valid. If **TRUE**, the context is still valid.

refresh

This operation may extend the useful lifetime of the SecurityContext. The precise behavior is implementation-specific. **refresh** may be called on both valid and expired contexts.

```
boolean refresh ();
```

Return Value

If the **refresh** operation returns a value of **FALSE**, then the context could not be refreshed. In this case, the caller should acquire a new context using the **Vault::init_security_context** interface. If **TRUE**, the context was successfully refreshed.

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 object, are sufficient to meet the access criteria for the requested operation. The interfaces are as follows.

access_allowed

```
interface AccessDecision {

    boolean access_allowed (
        in   SecurityLevel2::CredentialsList    cred_list,
        in   CORBA::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**.

Audit Objects

There are two types of audit objects:

- The audit decision object, used to find out whether an action needs to be audited. Similar audit decision objects are used for all audit policies.
- The audit channel objects, used by many of the implementation components (such as interceptors and security objects) and also used by applications to write audit records.

Audit Decision Objects

Audit Decision objects support the **`audit_needed`** interface defined in Section 15.5, Application Developer's Interfaces.

```
boolean         audit_needed (
    in AuditEventType                   event_type,
    in SelectorValueList                value_list
);
```

Parameters

event_type — The type of the event that has occurred.

selector_values — A list containing the values of the following audit selectors:

Initiator (the credentials-list of the principal whose action caused the event)

Object (the target object reference. If no target object exists, pass a reference to "self")

Operation (the name of the operation being invoked. Pass null if not applicable)

SuccessFailure (a boolean indicating whether the operation which triggered the event succeeded or failed)

Return Value

boolean — A return value of **TRUE** indicates that the event must be audited, otherwise **FALSE**.

A standard audit policy is proposed in Section 15.6, Administrator's Interfaces, 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 **`selector_values`** 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 AuditDecision object should obtain the current time itself.

Audit Channel Objects

Audit Channel objects support the **`audit_write`** interface defined in Section 15.5, Application Developer's Interfaces.

Principal Authentication

The Principal Authentication object defined in Section 15.5.3, Authentication of Principals, may also be called by implementation security objects, specifically the Vault.

Non-repudiation

The Non-repudiation services are accessible through the NRCredentials interface. Its functionality and operations are defined in Section 15.5, Application Developer's Interfaces.

15.7.5 Replaceable Security Services

It is possible to replace some security services independently of others.

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 Authentication, 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.

Replacing Access Decision Policies

Access control policies can be changed by replacing the Access Policy objects, which define and enforce access control policies (for example, substituting another Access

Policy object for DomainAccessPolicy). If a single object supports both AccessPolicy and *AccessDecision* interface, then only that object needs to be replaced. Otherwise, both AccessPolicy and AccessDecision objects may need to be replaced.

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.

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.

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.

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.

Linking to External Security Services

Most of an OMA-compliant secure system is unaware of the actual security services used, and that these may be shared with other systems. OMA-compliant secure system implementors are not required to make any interfaces other than those in Section 15.5, Application Developer's Interfaces, available to applications (though some implementations may expose more of the interfaces in this specification); ORBs and ORB interceptors use the interfaces specified in this section.

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 proposal 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 Section 15.4, Security Architecture, 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.

15.8 Security and Interoperability

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.

This section 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 in Section 15.8.5, DCE-CIOP with Security, 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.

15.8.1 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.
- The same security mechanism is used.

All other options build from this. The model for secure interoperability is shown in Figure 15-54.

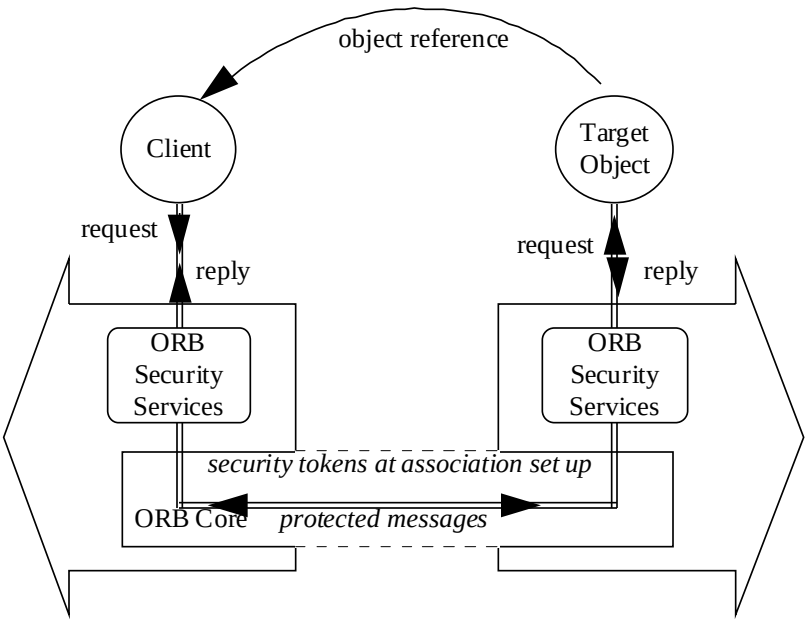

Figure 15-54 Secure Interoperability Model

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.

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. Therefore, 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 client's 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.

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.
- 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 is required.
- Some negotiation of security options for this mechanism is required, for example, the choice of cryptographic algorithms.

Protecting Messages

The invocation may be protected for integrity and/or confidentiality. In either case, the messages forming the request and reply are transformed by the ORB Security Services. For integrity, extra information (e.g., an integrity seal and sequence number) is added to the message so the target ORB Security Services can check that the message has not been changed and that no messages have been inserted or deleted in the sequence.

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.

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.

Security Mechanism Types

There are two major types of security mechanisms used in existing systems for security associations, which 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.
- Those using 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 that 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 interenterprise 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 that 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.

Interoperability where there is no mechanism in common is likely to be the subject of a future security RFP. It is expected that this would be done by a specialist interoperability *bridge* as described in the Security Interoperability Bridges section.

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 interoperate. 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.

Interoperating between Security Policy Domains

The previous sections 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:

- 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.
- The administrators of the two domains have agreed to 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.
- 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 on 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 could 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 in Section 15.7, Implementor's Security Interfaces, and this will probably call on a mechanism-specific Interdomain Service.

Secure Interoperability Bridges

Secure Interoperability Bridges between ORB domains are relevant to this architecture, as in the future, they may be specified as part of some secure CORBA-compliant systems. However, this specification does not describe how to build such bridges. If security interoperability bridges implemented separately from ORB Security Services are needed, they are expected to be the subject of separate RFPs.

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 reestablished using security services trusted by both client and target domains.

15.8.2 Protocol Enhancements

The following sections detail the enhancements required to the CORBA 2 interoperability specification for security.

- Section 15.8.3, CORBA Interoperable Object Reference with Security, defines the enhancements needed for the Interoperable Object Reference (IOR).
- Section 15.8.4, Secure Inter-ORB Protocol (SECIOP), defines the enhancements needed to secure GIOP messages, and Section 15.8.5, DCE-CIOP with Security, defines the DCE-CIOP with security.

15.8.3 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.

One profile thought necessary for OMG to define was a multicomponent profile, that is, a profile that itself consisted of tagged components. It is proposed that new multicomponent TAGs are defined, which allows the multicomponent profile to be used for IIOP.

However, use of tagged components within the multicomponent profile to carry IIOP, security, and other data may cause performance degradations in certain situations. For example, if an IOR carries many tagged components 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.

The following TAGs are defined:

- **IIOP components**, which can be used in a multicomponent profile (see Section B.7, Further Definition of ORB Interoperability).
- **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 an 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.

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. See the revised CORBA 2.0 specification (OMG Document Interop/96-05-01) for more information about placement of security information in IORs to support interoperable security in IIOP, GIOP protocols, DCE-CIOP, and ESIOP protocols via the multi-component profile. Chapter 10 of that document defines the IOR format, supported tags, and rules for composition of IOR components; Chapter 12 of that document describes the GIOP header and message formats and the IIOP IOR format; and Chapter 13 of that document describes the DCE-CIOP message formats.

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 be either encoded into the TAG value or in mechanism-specific data (see *Guidelines for Mechanism TAG Definition* in Appendix H, Interoperability Guidelines).

If this definition includes:

```
sequence <TaggedComponent> components;
```

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 multicomponent 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 multicomponent 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;
};
```

Parameters

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 15-9 Option Definitions

	target_supports	target_requires
NoProtection	The target supports unprotected messages	The target's minimal protection requirement is unprotected invocations
Integrity	The target supports integrity protected messages	The target requires messages to be integrity protected
Confidentiality	The target supports confidentiality protected invocation	The target requires invocations to be protected for confidentiality
DetectReplay	The target can detect replay of requests (and request fragments)	The target requires security associations to detect message replay
DetectMisordering	The target can detect sequence errors of requests and request fragments	The target requires security associations to detect message missequencing
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	The target requires establishment of trust in the client's identity

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.

Table 15-10 IOR Example

Tag	Value	Mech Specific Tag	Value
tag_sec_name	“Manchester branch”		
tag_association_ options	Supports and requires integrity to establish trust in the clients privileges		
tag_generic_sec_ mech	mech 1 oid	tag_sec_name	"MBnl"
		tag_association_ options	Supports and requires integrity, replay detection, misordering detection, to establish trust in the client’s security attributes
tag_generic_sec_ mech	mech 2 oid	tag_association_ options	Target requires and supports confidentiality, 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.

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 nonsecurity tagged components in the IOR multicomponent 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) that this target can use. The ORB selects a combination of interoperability protocol and a 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 that 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 the IOR table in Section 15.8.5, DCE-CIOP with Security, 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 **`override_default_QOP`** on the object reference.
- 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 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 15.5.6, Security Operations on Current). 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).

15.8.4 Secure Inter-ORB Protocol (SECIOP)

To provide a flexible means of securing interoperability between ORBs, a new protocol is introduced into the CORBA 2.0 Interoperability Architecture. This protocol sits below the GIOP protocol and provides a means of transmitting GIOP messages (or message fragments) securely.

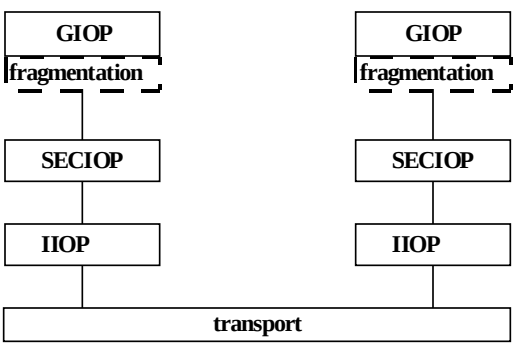

Figure 15-55 New CORBA 2.0 Protocol

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).

SECIOP Message Header

SECIOP messages share a common header format with GIOP messages defined in CORBA 2.0. The fields of this header have the following definition for SECIOP.

- 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 initial value for SECIOP is 1 major version, 0 minor version.
- 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.

A minor change is required to the GIOP header to rename the field GIOP_Version **`protocol_version`**.

SECIOP

The SECIOP protocol is used to control the secure association between clients and targets and provides a means for the transmission of protected messages between clients and targets.

Where possible, SECIOP 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).

Each name in the enumeration below corresponds to a structure discussed later in this section. The name of the designated structure is obtained by removing the initial "MT" from the name of the corresponding enumeration constant (for example, the structure corresponding to MTEstablishContext is named EstablishContext). The section titles under which the structures are discussed bear the names of the corresponding enumeration constants (i.e. the section names start with "MT").

SECIOP has the following message types:

```
enum MsgType {
    MTEstablishContext, MTCompleteEstablishContext,
    MTContinueEstablishContext, MTDiscardContext,
    MTMessageError, MTMessageInContext
};
struct ulonglong {
    unsigned long low;
    unsigned long high;
};
typedef ulonglong ContextId;
enum ContextIdDefn {
        Client, Peer, Sender
};
```

ContextId

This type is used to define the identifiers allocated by the client and target for the association.

ContextIdDefn

This enum is used to define the kind of context identifier held in a SECIOP message. The context identifier will either be the one specified by the client that established the context,