[2] The **Access Decision** object has the **access_allowed** operation as is used for enforcing access policies in the ORB (see below). The input parameters to this should normally specify:

- The privileges of the initiator of the action. The form of these depends on the specific policy. Some options are:
  - The privileges of the initiator as supplied by a **get_attributes** operation on **Current** (see Section 15.5.6.2, “The SecurityLevel1::Current Interface,” on page 15-99).
  - 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.

[3] The return value from the **access_allowed** operation is either **TRUE** signifying access is permitted, or **FALSE** signifying that it is not

[4] 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.

15.5.9.3 The SecurityLevel2::AccessDecision Interface

[1] The **Access Decision** object is a **locality constrained** object. The **AccessDecision** interfaces has the following single operation:

access_allowed

```
boolean access_allowed(
    in      SecurityLevel2::CredentialsList
                                    cred_list,
    in      Object                  target,
    in      CORBA::Identifier       operation_name,
    in      CORBA::Identifier       target_interface_name
);
```

Parameters

cred_list The list of **Credentials** associated with the request. The list may be empty (in the case of unauthenticated requests), it may contain only a single credential, or it may contain several credentials (in the case of delegated or otherwise cascaded requests). The **Access Decision** object is presumed to have rules for dealing with all these cases.

target The reference used to invoke the target object. The method invoked.

operation_name The name of the operation being invoked on the target.

target_interface_name The name of the interface to which the operation being invoked belongs. This may not be required in some implementations and

will only be required in cases in which the operation being invoked does not belong to the interface of which the target object is a direct instance.

Return Value

boolean A return value of **TRUE** indicates that the request should be allowed, otherwise **FALSE**.

15.5.9.4 Portability Implications

[1] 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.).

[2] 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.

[3] 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 Delegation Facilities

15.5.10.1 Description of Facilities

[1] An operation on a target object may result in calls on many other objects as described in Section 15.3.6, "Delegation," on page 15-27. 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.

[2] 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.

[3] 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.

- **application access** policy (**Security::SecApplicationAccess**, interface **SecurityAdmin::AccessPolicy**), which applications may use to manage and enforce their access policies.
- **application audit** policy (**Security::SecApplicationAudit**, interface **SecurityAdmin::AuditPolicy**), which applications can use to manage and enforce their audit policies.
- **non-repudiation** policies (**Security::SecNonRepudiation**, interface **SecurityAdmin::NRPolicy**) determine the rules for the generation and use of evidence.

[4] There is also a policy concerned with creation of object references, which is enforced by **POA::create_reference** and variants thereof or equivalent operation. This is the **construction policy** (**CORBA::SecConstruction**) which controls whether a new domain is created when an object of a specified type is created.(See CORBA V2.2 Section 4.9)

[5] 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.

[6] Operations are provided for setting all the types of security policies previously listed. In each case, these management operations permit administration of standard policy semantics supported by the interfaces defined in this specification. It is also possible for implementors to replace the policy objects, the operations of which are defined in this specification, with different policy objects supporting different semantics; in general such policy objects will also have management operations that are different from those defined in this specification.

15.6.4 Access Policies

[1] There are two types of invocation access policies: the Client Invocation Access policy (**Security::SecClientInvocationAccess**), which is used at the client side of an invocation, and the Target Invocation Access policy (**Security::SecTargetInvocationAccess**), which is used at the target side.

[2] There is one policy type for application access. However, no standard administrative interface to this is specified, as different applications have different requirements.

[3] 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,” on page 15-83); rights are described in the next section.

15.6.4.1 Rights

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

[2] In rights-based systems, **Access Policy** objects *grant* rights to PrivilegeAttributes; for each operation in the interface of a secure object, some set of rights is *required*. Callers must be granted these required rights in order to be allowed to invoke the operation.

[3] 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.

[4] A **Required Rights** object is available as an attribute of **Current** in every execution context. Every **Required Rights** object will get and set the same information, so it does not matter which instance of the **RequiredRights** interface is used. The required rights for all operations of all secured interfaces are assumed to be accessible through any instance of **RequiredRights**.

[5] Note that required rights are characteristics of interfaces, *not* of instances. All instances of an interface, therefore, will always have the same required rights.

[6] 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

[7] This specification provides a standard set of rights for use with the **DomainAccessPolicy** interface defined later in this section. These rights may not satisfy all access control requirements. However; to allow for extensibility, rights are grouped into Rights Families. The **RightsFamily** containing the standard rights is called "**corba**," and contains four rights: "**g**" (interpreted to mean "**get**"), "**s**" (interpreted to mean "**set**"), "**m**" (interpreted to mean "**manage**") and "**u**" (interpreted to mean "**use**"). Implementations may define additional Rights Families. **Rights** are always qualified by the **RightsFamily** to which they belong.

15.6.4.2 The SecurityLevel2::RequiredRights Interface

[8] A **Required Rights** object can be thought of as a table; an example Required Rights table appears later in this section. Note that implementations need not manage required rights on an interface-by-interface basis; **Required Rights** objects should be thought of as databases of policy information, in the same way as Interface Repositories are databases of interface information. Thus in many implementations, all calls to the **RequiredRights** interface will be handled by a single Required Rights object instance, or by one of a number of replicated instances of a master Required Rights object instance.

[9] An operation's entry in the Required Rights table lists a set of rights, qualified (or "tagged") as usual with the **RightsFamily**. It also specifies a *Rights Combinator*; the rights combinator defines how entries with more than one required right should be interpreted. This specification defines two Rights Combinators: *AllRights* (which means that all rights in the entry must be granted in order for access to be allowed), and *AnyRight* (which means that if any right in the entry is granted, access will be allowed).

[10] 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

[11] This operation retrieves the rights required to execute the operation specified by **operation_name** of the interface specified by **obj**. **obj**'s interface will be determined and used to retrieve required rights. The returned values are a list of rights and a combinator describing how the list of rights should be interpreted if it contains more than one entry.

```
void get_required_rights(
    in      Object              obj,
    in      CORBA::Identifier   operation_name,
    in      CORBA::RepositoryId interface_name,
    out     RightsList          rights,
    out     RightsCombinator    rights_combinator
);
```

Parameters

obj The object for which required rights are to be returned.

operation_name The name of the operation for which required rights are to be returned.

interface_name The name of the interface in which the operation described by **operation_name** is defined, if this is different from the interface of which obj is a direct instance. Not all implementations will require this parameter; consult your implementation documentation.

rights The returned list of required rights.

rights_combinator The returned rights combinator.

Return Value

None.

set_required_rights

[12] This operation updates the rights required to execute the operation specified by **operation_name** of the interface specified by **interface_name**. The caller must provide a list of rights and a combinator describing how the list of rights should be interpreted if it contains more than one entry. Note that consistency issues arising from replication of **Required Rights** objects or distribution of the **RequiredRights** interface must be handled correctly by implementations; after a call to **set_required_rights** changes an interface's required rights, all subsequent calls to **get_required_rights**, from any client, must return the updated rights set.

```
void set_required_rights(
    in      string              operation_name,
    in      CORBA::RepositoryId interface_name,
    in      RightsList          rights,
    in      RightsCombinator    rights_combinator
);
```

Parameters

operation_name	The name of the operation for which required rights are to be updated.
interface_name	The name of the interface whose required rights are to be updated.
rights	The desired new list of required rights.
rights_combinator	The desired new **RightsCombinator**.

Return Value

None.

15.6.4.3 The SecurityAdmin::AccessPolicy Interface

[1] This is the root interface for the various kinds of invocation access control policy. This interface supports querying of the effective access granted by a set of attributes by an invocation access policy. It inherits the **CORBA::Policy** interface and has a single operation, **get_effective_rights**.

get_effective_rights

[2] This operation returns the current effective rights (of family **RightsFamily**) granted by this **Access Policy** object to the subject possessing all privilege attributes in the list of attributes **attrib_list**.

```
RightsList get_effective_rights(
    in      AttributeList           attrib_list,
    in      ExtensibleFamily        rights_family
);
```

Parameters

attrib_list	A list of attributes obtained from one or more **Credentials** using the **get_attributes** operation.
rights_family	The family of rights that is desired as return value.

Return Value

A list of effective rights that are consistent with the **attrib_list** and the access policy, of the family specified by **rights_family**

[3] Note that this specification does not define how an **Access Policy** object combines rights granted through different Privilege Attribute entries, in case a subject has more than one Privilege Attribute to which the Access Policy grants rights. However, this call will cause the **Access Policy** object to combine rights granted to all privilege attributes in the input **AttributeList** (using whatever operation it has implemented), and return the result of the combination.

[4] **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.

15.6.4.4 Specific Invocation Access Policies

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

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

15.6.4.5 The Domain AccessPolicy Object

[1] The **Domain Access Policy** object with the **SecurityAdmin::DomainAccessPolicy** interface provides discretionary access policy management semantics. CORBA implementations with policy requirements, which cannot be met by the **Domain Access Policy** abstraction, may choose to implement different **Access Policy** objects; for example, they may choose to implement access control policy management using capabilities.

Domains

[2] 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.

[3] A **Domain Access Policy** object defines the access policy, which grants a set of named "subjects" (e.g. users), a specified set of "rights" (e.g. **g**, **s**, **m**, **u**) to perform operations on the "objects" in the domain. A Domain Access Policy can be represented by a table whose row labels are the names of subjects, and whose cells are filled with the rights granted to the subject named in that row's label, as in Table 15-2 (note that the use of the Delegation State will be discussed in the section of the same name next).

Table 15-2 DomainAccessPolicy

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

[4] This **Domain Access Policy** grants the rights "**g**" and "**s**" to Alice and Zeke, and the right "**g**" to Bob and Cathy. (The annotation "**corba**" prefixing the granted rights indicates which Rights Family, as defined in the previous section, each of the rights in the table is drawn from. In this case, all rights are drawn from Domain Access Policy's standard "**corba**" Rights Family. The delegation state column is described under the heading "Delegation States".)

Domain Access Policy Use of Privilege Attributes

[5] Administration of principals by individual identity is costly, so the Domain Access Policy aggregates principals for access control. A common aggregation is called a "user group." This specification generalizes the way users are aggregated, using

"Privilege Attributes" (as defined in Section 15.3.4.3, "Access Policies," on page 15-23). Users may have many kinds of privilege attributes, including groups, roles, and clearances (note that user access identities, often referred to simply as "user identities" or "userids," are considered to be a special case of privilege attributes). The **Domain Access Policy** object uses Privilege Attributes as its subject entries.

[6] 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 15-3 User Privilege Attributes (Not Defined by This Specification)

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

[7] Given the definitions in this table, we can simplify our Domain Access Policy as follows (note that, for convenience, each PrivilegeAttribute entry is annotated in the table with its **PrivilegeAttribute** type).

Table 15-4 Domain Access Policy (with Privilege Attributes)

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

Delegation State

[8] The **Domain Access Policy** abstraction allows administrators to grant different rights when a Privilege attribute is used by a delegate than those granted to the same Privilege attribute when used by an initiator (note that "initiator" means the principal issuing the first call in a delegated call chain; that is, the only client in the call chain that is not also a target object). The **Domain Access Policy** shown next illustrates the use of this feature.

Table 15-5 Domain Access Policy (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**--

[9] This **Domain Access Policy** grants Alice the "**g**" and "**s**" rights when she accesses an object as an initiator, but only the "**g**" right when a delegate using her identity accesses the same object.

Domain Access Policy Use of Rights and Rights Families

[10] The rights granted to a Privilege Attribute by a **Domain Access Policy** entry must each be "tagged" with the RightsFamily to which they belong; each **Domain Access Policy** entry can grant its row's **PrivilegeAttribute** rights from any number of different Rights Families.

[11] Implementations may define new Rights Families in addition to the standard "**corba**" family, though this should be done only if absolutely necessary, since new Rights Families complicate the administrator's model of the system.

Access Decision Use of AccessPolicy and RequiredRights

[12] The **Access Decision** object is described in Section 15.5.9.2, "The Access Decision Object," on page 15-108. It is used at run-time to perform access control checks. **Access Decision** objects rely upon **Access Policy** objects to provide the policy information upon which their decisions are based.

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

Table 15-6 Interface Instances

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

[14] The **Domain Access Policy** object illustrated next has been updated to include a list of rights of type "other" granted to each of the Privilege attributes.

Table 15-7 Domain Access Policy (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: ------

[15] Table 15-8 shows Required Rights for three object interfaces (c1, c2, and c3), using the standard Rights Family "**corba**" and a second Rights Family, "other," whose rights set is assumed to be {g, u, o, m, t, s}.

Table 15-8 Required Rights for Interfaces c1, c2 and c3

Required Rights	Rights Combinator	Operation	Interface
corba:s	all	m1	c1
corba:gs	any	m2	
other:u	all	m3	c2
other:ms	all	m4	
other: s	all	m5	c3
corba:gs	all	m6	

[16] Using this, we can calculate the effective access granted by this Domain Access Policy.

- alice can execute operations m1 and m2 of objects obj_1, obj_8, and obj_n as an initiator, but may execute only m2 as a delegate.
- alice can execute operations m3 and m4 of objects obj_2, and obj_5 as an initiator, but may execute no operations of obj_2 and obj_5 as a delegate.
- alice can execute operations m5 and m6 of object obj_12 as an initiator, but may execute no operations as a delegate.
- “programmers” can execute operation m2 of objects obj_1, obj_8, and obj_n as an initiator, but no operations as a delegate.
- “programmers” can execute operation m3 of objects obj_2 and obj_5 as an initiator, but no operations as a delegate.
- “administrators” can execute operations m1 and m2 of objects obj_1, obj_8, and obj_n as an initiator, but no operations as a delegate.
- “administrators” can execute operations m5 and m6 of object obj_12 as an initiator, but no operations as a delegate.

15.6.4.6 The SecurityAdmin::DomainAccessPolicy Interface

[1] The **Domain Access Policy** object provides operations for managing access policy through the **DomainAccessPolicy** interface.

[2] Each domain manager may have at most one **Access Policy** object, and therefore at most one **Domain Access Policy** (though an object instance may have more than one domain manager, and therefore, more than one **Domain Access Policy**). The **DomainAccessPolicy** interface inherits the **AccessPolicy** interface and defines operations to specify which subjects can have which rights as follows.

grant_rights

[3] This operation grants the specified **rights** to the privilege attribute **priv_attr** in delegation state **del_state**.

[4] 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
);
```

Parameters

priv_attr	privilege attributes to be affected
del_state	delegation state to be set
rights_family	the family of rights to be affected
rights	the list of rights to be granted

Return Value

None.

revoke_rights

[5] This operation revokes the specified **rights** of the privilege attribute **priv_attr** in delegation state **del_state**.

[6] 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
);
```

Parameters

priv_attr	privilege attributes to be affected
del_state	delegation state to be set
rights_family	the family of rights to be affected
rights	the list of rights to be revoked

Return Value

None.

replace_rights

[7] This operation replaces the current rights of the privilege attribute **priv_attr** in delegation state **del_state** with the **rights** provided as input.

[8] 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
);
```

Parameters

priv_attr	privilege attributes to be affected
del_state	delegation state to be set
rights_family	the family of rights to be affected
rights	the list of rights to be replaced

Return Value

None.

get_rights

[9] This operation returns the current rights (of type **RightsList**) of the privilege attribute **priv_attr** in delegation state **del_state**.

[10] Utilities that manage access policy should use this operation to retrieve rights granted to an individual privilege attribute.

```
RightsList get_rights(
    in    Attribute          priv_attr,
    in    DelegationState    del_state,
    in    ExtensibleFamily   rights_family
);
```

Parameters

priv_attr	privilege attributes to which the requested rights are granted
del_state	delegation state to be set
rights_family	the family of rights to be retrieved

Return Value

a list of rights granted to the specified privilege attribute of the specified rights family in the specified delegations tate.

15.6.5 Audit Policies

[1] There are two invocation audit policies: the **SecClientInvocationAudit** policy, which is used at the client side of an invocation, and the **SecTargetInvocationAudit** policy, which is used at the target side. There is also an application audit policy type.

[2] 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.