Figure 12.9
Authentication Interface of CORBA Security Service

```
interface Credentials {
 Credentials copy();
 void destroy();
 ...
 boolean set_privileges(in boolean force_commit,
                        in AttributeList requested_privileges,
                        out AttributeList actual_privileges);
 AttributeList get_attributes (in AttributeTypeList attributes);
 boolean is_valid (out UtcT expiry_time);
 boolean refresh();
};

interface PrincipalAuthenticator {
 AuthenticationStatus authenticate(
                        in AuthenticationMethod method,
                        in SecurityName security_name,
                        in Opaque auth_data,
                        in AttributeList privileges,
                        out Credentials creds,
                        out Opaque continuation_data,
                        out Opaque auth_specific_data);
 ...
}
```

use of the `Current` object when we discussed distributed transactions. In addition to transaction contexts, the `Current` object also stores the credentials of the principal on whose behalf the session is executed and they are then used for access control purposes.

12.4.2 Access Control

Figure 12.10 shows the interfaces that are available for the management of access rights in credentials. These interfaces are used to construct administration tools so that administrators can set and revoke access rights for individual principals. The definition of access rights is based on *privileges*. Privileges are assigned to a principal's credentials object using `set_privileges` shown in Figure 12.9. Access to this operation is usually restricted to an administrator. The `DomainAccessPolicy` interface allows an administrative tool to grant, revoke, replace and get the set of access rights that are defined for a privilege.

In order for an administrator to effectively control the access to objects, the CORBA Security service defines the `RequiredRights`. CORBA performs access control at a type-level of abstraction, which means that it is not possible to define different access rights for different instances of a type. `RequiredRights` enables an administrator to associate a set of rights with an operation that is part of an interface and demand that any principal wishing to execute that operation must possess those rights. A `RightsCombinator` defines how sequences of more than one required right are determined.

```
interface DomainAccessPolicy : AccessPolicy {
 void grant_rights(in SecAttribute priv_attr,
                   in DelegationState del_state,
                   in ExtensibleFamily rights_family,
                   in RightsList rights);
 void revoke_rights(in SecAttribute priv_attr,
                    in DelegationState del_state,
                    in ExtensibleFamily rights_family,
                    in RightsList rights);
 void replace_rights (in SecAttribute priv_attr,
                      in DelegationState del_state,
                      in ExtensibleFamily rights_family,
                      in RightsList rights);
 RightsList get_rights (in SecAttribute priv_attr,
                        in DelegationState del_state,
                        in ExtensibleFamily rights_family);
};
```

Figure 12.10
CORBA Interfaces for Administering Credentials

```
// Declarations related to Rights
struct ExtensibleFamily {
 unsigned short family_definer;
 unsigned short family;
};
struct Right {
 ExtensibleFamily rights_family;
 string right;
};
typedef sequence <Right> RightsList;

interface RequiredRights{
 ...
 void set_required_rights(
                        in string operation_name,
                        in CORBA::RepositoryId interface_name,
                        in RightsList rights,
                        in RightsCombinator rights_combinator);
};
```

Figure 12.11
CORBA Access Control Interface for Administrators

To summarize the administration of access rights we note that access rights are defined in a three-stage process. First, credential objects capture the assignment of privileges to principals. Secondly, access policies capture the access rights that are granted to privileges. Finally, for each operation of each interface, required rights define the rights that principals must hold (through their privileges) in order to be able to use the operation.

CORBA access rights have three parts: credentials, access policies, and access rights.

Figure 12.12 shows the rather simple client interface to access control. A client can test whether the credentials of the principal currently using the client are sufficient to execute a particular operation in a particular interface.

Figure 12.12
CORBA Access Control Interface for Clients

```
typedef sequence <Credentials> CredentialsList;

interface AccessDecision {
   boolean access_allowed(
                    in CredentialsList cred_list,
                    in Object target,
                    in CORBA::Identifier operation_name,
                    in CORBA::Identifier target_interface_name);
};
```

12.4.3 Non-Repudiation

The essential operations for the client interface to the non-repudiation interfaces of the CORBA Security service are shown in Figure 12.13. The `NRCredential` supports the creation of a non-repudiation token that represents a particular type of evidence. The client provides the data that is needed for the generation of the evidence in `input_buffer` and determines the type of evidence that is to be generated. The client then receives a token that

Figure 12.13
CORBA Non-Repudiation Interface

```
enum EvidenceType {
 SecProofofCreation,
 SecProofofReceipt,
 SecProofofApproval,
 SecProofofRetrieval, ...
};

interface NRCredentials : Credentials {
 ...
 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);

 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);
 ...
};
```

it can record or pass on whenever evidence is necessary. Servers or adjudicators can use the evidence token in an evidence check to retrieve the original evidence data and a verification that the token was generated.

The CORBA Security service further includes a number of interfaces that support the construction of tools for the administration of non-repudiation policies. These rules determine the involvement of trusted third parties in the evidence generation, the roles in which they may be involved and the duration for which the generated evidence is valid. Moreover, policies may determine rules for adjudication, which for example determine the authorities that may be used for the adjudication of disputes.

12.4.4 Auditing

Auditing of system events relevant to security is transparent for programmers of client and server objects. The `AuditPolicy` interface in Figure 12.14 shows CORBA interfaces that are used by administrative tools for the definition of system auditing policies.

Security auditing can be kept transparent for CORBA server and client programmers.

Figure 12.14
CORBA Interfaces for Administering Auditing Policies

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

typedef unsigned long AuditChannelId;
typedef unsigned short EventType;
const EventType AuditAll = 0;
const EventType AuditPrincipalAuth = 1;
const EventType AuditSessionAuth = 2;
const EventType AuditAuthorization = 3;
const EventType AuditInvocation = 4;
const EventType AuditSecEnvChange = 5;
...
struct AuditEventType {
   ExtensibleFamily event_family;
   EventType event_type;
};

typedef sequence <AuditEventType> AuditEventTypeList;

interface AuditPolicy : CORBA::Policy {
 void set_audit_selectors (
        in CORBA::InterfaceDef object_type,
        in AuditEventTypeList events,
        in SelectorValueList selectors);
 ...
 void set_audit_channel (
        in AuditChannelId audit_channel_id);
};
```