Common Secure Interoperability V2 Specification

OMG Adopted Specification

Change History

CSIv2-011701 Draft Adopted Specification

The Adopted Submission was modified in form for inclusion as chapter 16 in the CORBA v2.4 specification.

The following Issues were resolved editorially:

- Issue 3932: Service ID missing (csiv2-ftf)
- Issue 3974: CSIv2 IOR Tagged component Identifier values (csiv2-ftf)
- Issue 3975: INVALID recommended cipher suites (csiv2-ftf)

Change bars are relative to the adopted submission.

Secure Interoperability 16

| [1] This chapter defines the CORBA Security Attribute Service (SAS) protocol and its use within the CSIv2 architecture to address the requirements of CORBA security for interoperable authentication, delegation, and privileges.

Contents

[2] This chapter contains the following sections.

Guide to Chapter

- Section 16.1, "Overview," describes the Security Attribute Service (SAS) protocol and lists the assumptions under which the protocol was designed.
- Section 16.2, "Protocol Message Definitions," defines the SAS protocol message definitions and their token formats.

- Section 16.3, “Security Attribute Service Protocol,” defines the protocol, using state tables and explanations of various scenarios. The chapter defines rules by which clients and targets should exchange and interpret security attribute contexts.
- Section 16.4, “Transport Security Mechanisms,” discusses secure transports.
- Section 16.5, “Interoperable Object References,” specifies the definition and semantics of the CSIv2 tagged components that are put into the standard Interoperable Object Reference (IOR).
- Section 16.6, “Conformance Levels,” identifies the minimum level of functionality that is required to conform to the CSIv2 protocol (Conformance Level 0) and defines additional conformance levels that support more advanced functionality.
- Section 16.7, “Sample Message Flows and Scenarios,” describes scenarios and provides sample message flows.
- Section 16.8, “References for this Chapter,” lists other specifications on which this chapter depends.
- Section 16.9, “IDL,” defines the concrete syntax of the message formats that correspond to the SAS protocol and the corresponding tag components used in IORs.

16.1 Overview

[3] The SAS protocol is designed to exchange its protocol elements in the service context of GIOP request and reply messages that are communicated over a connection-based transport. The protocol is intended to be used in environments where transport layer security, such as that available via SSL/TLS or SECIOP, is used to provide message protection (that is, integrity and or confidentiality) and server-to-client authentication. The protocol provides client authentication, delegation, and privilege functionality that may be applied to overcome corresponding deficiencies in an underlying transport.[1] The SAS protocol facilitates interoperability by serving as the higher-level protocol under which secure transports may be unified.

[4] The SAS protocol is divided into two layers:

- The authentication layer is used to perform client authentication where sufficient authentication could not be accomplished in the transport.
- The attribute layer may be used by a client to push (that is, deliver) security attributes (identity and privilege) to a target where they may be applied in access control decisions.

[5] The attribute layer also provides a means for a client to assert identity attributes that differ from the client’s authentication identity (as established in the transport and/or SAS authentication layers). This identity assertion capability is the foundation of a general-purpose impersonation mechanism that makes it possible for an intermediate to

1. For example, the SSL/TLS protocol does not enforce client authentication. Moreover, in a given environment, certificate-based client authentication may not be feasible because clients often do not have a certificate.

act on behalf of some identity other than itself. An intermediate's authority to act on behalf of another identity may be based on trust by the target in the intermediate, or on trust by the target in a privilege authority that endorses the intermediate to act as proxy for the asserted identity. Identity assertion may be used by an intermediate to assume the identity of its callers in its calls.

[6] The SAS protocol is modeled after the Generic Security Service API (GSSAPI) token exchange paradigm. A client initiates a context exchange by including a protocol element in the service context of its request that instructs the target to initiate a security context. The target either rejects or accepts the context.[2] When a target rejects a context, the target will reject the request and return an exception that contains a SAS protocol element that identifies the reason the context was rejected. When a target accepts a context, the reply to the request will carry a SAS protocol element that indicates that the context was accepted.

[7] The SAS protocol element sent to initiate a security context carries layer-specific security tokens as necessary to establish the SAS authentication-layer and attribute-layer functionality corresponding to the context. Standard token formats are employed to represent the layer-specific authentication and attribute tokens. If the context includes SAS authentication-layer functionality, the protocol element will contain a mechanism-specific GSSAPI initial context token that authenticates the client to the target. If the context includes attribute-layer privilege attributes (and possibly proxy endorsements), they will be contained in an attribute certificate signed by a privilege authority and corresponding to the subject of the invocation. If the context includes an attribute-layer identity assertion, the asserted identity will be represented in a standard name form corresponding to the technology domain of the asserted identity.

[8] The SAS protocol supports the establishment of both transient and reusable security contexts. Transient contexts, also known as stateless contexts, exist only for the duration of the GIOP request that was used to establish the context. Reusable contexts, also known as stateful contexts, endure until they are discarded, and can be referenced for use with subsequent requests. The SAS protocol includes a simple negotiation protocol that defines a least-common-denominator form of interoperability between implementations that support only transient contexts and those that support both transient and reusable forms.

16.1.1 Assumptions

[9] The SAS protocol was designed under the following assumptions:

- Secure interoperability is predicated on the use of a common transport-layer security mechanism, such as that provided by SSL/TLS.[3]
- The transport layer provides message protection as necessary to protect GIOP input and output request arguments.

2. In the GSSAPI protocol, a target can challenge a client for additional context-establishment information. This is not true of the SAS context protocol, which assumes that at most one message in each direction may be used to establish a context.
3. Transport security mechanisms include unprotected transports within trusted environments.

- The transport layer provides target-to-client authentication as necessary to identify the target for the purpose of ensuring that the target is the intended target.
- Transport-layer security can ensure that the client does not have to issue a preliminary request to establish a confidential association with the intended target.[4]
- To support clients that cannot authenticate using transport-layer security mechanisms, the SAS protocol shall provide for client authentication above the transport layer.
- To support the formation of security contexts using GIOP service context, the SAS protocol shall require at most one message in each direction to establish a security context.
- The protocol shall support security contexts that exist only for the duration of a single request/reply pair.
- The protocol shall support security contexts that can be reused for multiple request/reply pairs.
- Targets cannot rely on clients to manage the lifecycle of reusable security contexts accepted by the target.
- Clients that reuse security contexts shall be capable of processing replies that indicate that the context has been discarded by the target.

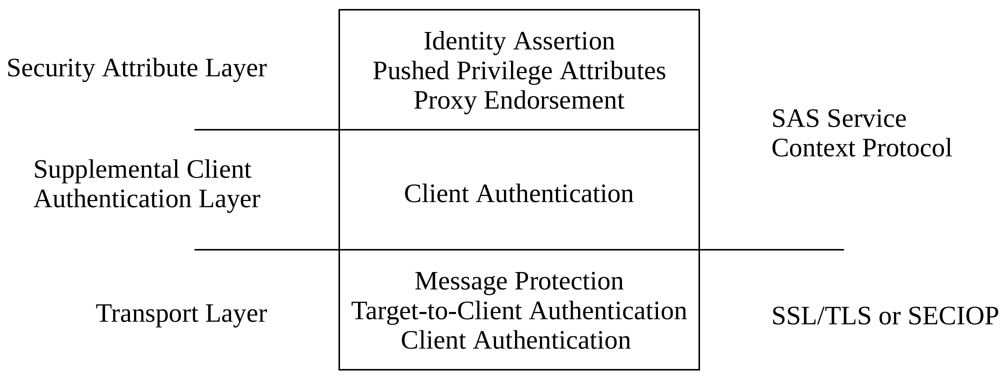


Figure 16-1 CSIv2 Security Architecture

4. This assumption does not preclude the use of such mechanisms, but rather sustains the use of this protocol in environments where such mechanisms are not considered favorably.

16.2 Protocol Message Definitions

16.2.1 The Security Attribute Service Context Element

[10] This specification defines a new GIOP service context element type, the security attribute service (SAS) element.

[11] The SAS context element may be used to associate any or all of the following contexts with GIOP request and reply messages:

- Identity context, to be accepted based on trust
- Authorization context, including authorization-based delegation context
- Client authentication context

[12] A new **context_id** has been defined for the SAS element.

```
const ServiceId SecurityAttributeService = 15;
```

[13] The **context_data** of a SAS element is an encapsulation octet stream containing a SAS message body marshalled according to the CDR encoding rules. The formats of the SAS message bodies are defined in the next section.

```
struct ServiceContext {
   ServiceId context_id;
   sequence <octet> context_data;
};
```

[14] At most one instance of this new service context element may be included in a GIOP request or reply.

16.2.2 SAS context_data Message Body Types

[15] Four message types comprise the security attribute service context management protocol. Each security attribute service context element shall contain a message body that carries one of the following message body types:

- **EstablishContext**

  Sent by a client security service (CSS) to establish a security attribute service context.

- **ContextError**

  Sent by a target security service (TSS) to indicate errors that were encountered in context creation, in the message protocol, or in use of a context.

- **CompleteEstablishContext**

  Sent by a target security service (TSS) to indicate the outcome of a successful request to establish a security attribute service context.

- **MessageInContext**

  Sent by a client security service (CSS) to associate request messages with an existing stateful security attribute service context. This message may also be used to indicate that the context should be discarded after processing the request.

  Stateful contexts, also known as reusable contexts, endure until they are discarded, and can be referenced for use with subsequent requests.

[16] A client security service (CSS) is the security service associated with the ORB that is used by the client to invoke the target object. A target security service (TSS) is the security service associated with the ORB that hosts the target object.

EstablishContext Message Format

[17] An EstablishContext message is sent by a CSS to establish a SAS context with a TSS. The SAS context and the context identifier allocated by the CSS to refer to it are scoped to the transport layer connection or association over which the CSS and TSS are communicating. When an association is dismantled, all SAS contexts scoped to the connection shall be invalidated and may be discarded. The EstablishContext message contains the following fields:

- **client_context_id**

  The CSS allocated identifier for the security attribute service context. A stateless CSS shall set the client_context_id to 0, indicating to the TSS that it is stateless. A stateful CSS may allocate a nonzero client_context_id. See Section , “Stateful/Reusable Contexts,” on page 16-21 for a definition of the rules governing the use and allocation of context identifiers.

- **authorization_token**

  May be used by a CSS to “push” privilege information to a TSS. A CSS may use this token to send proxy privileges to a TSS as a means to enable the target to issue calls as the client.

- **identity_token**

  Carries a representation of the invocation identity for the call (that is, the identity under which the call is to be authorized). The **identity_token** carries a representation of the invocation identity in one of the following forms:

  - A typed mechanism-specific representation of a principal name
  - A chain of identity certificates representing the subject and a chain of verifying authorities
  - A distinguished name
  - The anonymous principal identity (a type, not a name)

  An **identity_token** is used to assert a caller identity when that identity differs from the identity proven by authentication in the authentication layer(s). If the caller identity is intended to be the same as that established in the authentication layer(s), then it does not need to be asserted in an **identity_token**.

- **client_authentication_token**

  Carries a mechanism-specific GSS initial context token that authenticates the client to the TSS. It contains a mechanism type identifier and the mechanism-specific evidence (that is, the authenticator) required by the TSS to authenticate the client.

  When an initial context token contains private credentials, such as a password, this message may be safely sent only after a confidential connection with a trusted TSS has been established. The determination of when it is safe to send a client authentication token in an EstablishContext message shall be considered in the context of the CORBA location-binding paradigm for persistent objects (where an invocation may be "location forwarded" by a location daemon to the target object). This issue is considered in Section 16.5.3, "Client-Side Requirements and Location Binding," on page 16-41.

[18] When a TSS is unable to validate a security attribute service context, the TSS shall not dispatch on the target object method invocation. The TSS shall reply with a ContextError message that carries major and minor codes indicating the reason for the failure.

[19] If an EstablishContext message contains an identity token, then it is the responsibility of the TSS to extract a principal identity from the identity token and determine if the identity established in the authentication layer(s) is trusted to assert the extracted identity. If so, the asserted identity is used as the caller identity in the target's authorization determination.

[20] The processing of a request to establish a context that arrives on a one-way call shall be the same as an ordinary call, except that the TSS will not send an indication of the success (CompleteEstablishContext) or failure (ContextError) of the context validation.

ContextError Message Format

[21] A ContextError message is sent by a TSS in response to an EstablishContext or MessageInContext message to indicate to the client that the TSS detected an error. Section 16.3.4, "CSS State Machine," on page 16-25 defines the circumstances under which a TSS returns specific error values and exceptions. The ContextError message contains the following fields:

- **client_context_id**

  The value of the **client_context_id** that identifies the CSS context in the EstablishContext or MessageInContext message in response to which the ContextError is being returned.

- **major_status**

  The reason the TSS rejected the context.

- **minor_status**

  A more specific error code that further defines the reason for rejection in the context of the major status.

- **error_token**

  A GSS mechanism-specific error token. When an EstablishContext message is rejected because it contains a **client_authentication_token** (a GSS initial context token) that is invalidated by the TSS, then depending on the mechanism, the TSS may return a mechanism-specific GSS error token in this field. Not all GSS mechanisms produce error tokens in response to initial context token validation failures.

[22] In all circumstances where a TSS returns a ContextError, the GIOP request that carried the rejected SAS context shall not be dispatched by the target ORB.

CompleteEstablishContext Message Format

[23] A CompleteEstablishContext message is sent by a TSS in response to an EstablishContext message to indicate that the context was established. The CompleteEstablishContext message contains the following fields:

- **client_context_id**

  The CSS allocated identifier for the security attribute context. It is returned by the target so that a stateful CSS can link this message to the EstablishContext request. A TSS shall always return the value of the **client_context_id** it received in the EstablishContext message.

- **context_stateful**

  The value returned by the TSS to indicate whether or not the established context is stateful, and thus reusable. A stateless TSS shall always return false. A stateful TSS shall return true if the established context is reusable. Otherwise a stateful TSS shall return false.

- **final_context_token**

  The GSS mechanism-specific final context token that is returned by a TSS if the client requests mutual authentication. When a TSS accepts an EstablishContext message containing an initial context token that requires mutual authentication, the TSS shall return a mechanism-specific final context token. Not all GSS mechanisms support mutual authentication, and thus not all responses to initial context tokens may include final (or output) context tokens.[5]

  When a CompleteEstablishContext message contains a **final_context_token**, the token shall be applied (with GSS_Init_sec_context) to the client-side GSS state machine.

[24] Two or more stateful SAS contexts are equivalent if they are established over the same transport layer connection or association, have the same non-zero **client_context_id** and have byte-equivalent identity, authorization, and authentication tokens.

5. SAS layer authentication capabilities are designed to authenticate client to server where such authentication did not occur in the transport. The SAS protocol is predicated on server-to-client authentication having occurred in the transport layer, and in advance of the request. Server-to-client authentication in service context (which requires that the target return a **final_context_token**) is not the typical use model for SAS layer authentication capabilities.

[25] A multithreaded CSS may issue multiple concurrent requests to establish (that is, with an EstablishContext message) an equivalent stateful SAS context.

[26] A TSS shall not create a duplicate stateful SAS context in response to a request to establish a context that is equivalent to an existing context.

[27] A TSS shall return an exception containing a ContextError service context element if it receives a stateful EstablishContext message with a **client_context_id** that matches that of an existing context (established over the same transport layer connection or association) and for which any of the security tokens arriving in the message are not byte-equivalent to those recorded in the existing context. The request shall also be rejected. The exception and error values to be returned are defined in Section 16.3.4, "CSS State Machine," on page 16-25.

Table 16-1 CompleteEstablishContext Message Semantics

client_context_id in EstablishContext Message	**client_context_id in CompleteEstablishContext Message**	**context_stateful in CompleteEstablishContext Message**	**Semantic**
0	0	False	Client requested stateless context
N != 0	N	False	TSS is stateless or TSS did not choose to remember context. In either case, if the client attempts to reuse the context (via MessageInContext) it should expect to receive an error
		True	Stateful TSS accepted reusable context

MessageInContext Message Format

[28] A MessageInContext message is used by a CSS that wishes to reuse an existing context with a request. A CSS may also use this message to release context that it has established with a stateful TSS. The MessageInContext message contains the following fields:

- **client_context_id**

  The nonzero context identifier allocated by the client in the EstablishContext message used to create the context.

- **discard_context**

  A boolean value that indicates whether the CSS wishes the TSS to discard the context after it processes the request. A value of true indicates that the CSS wishes the context to be discarded, a value of false, indicates that it does not. The purpose of the **discard_context** field is to allow a CSS to help a TSS manage the cleanup of reusable contexts.[6]

6. Stateful clients are under no obligation to manage TSS state, so their use of this message for that purpose is discretionary.

[29] Any request message may be used to carry a MessageInContext message to a target. A TSS that receives a MessageInContext message shall complete the processing of the request before it discards the context (if discard_context is set to true).

[30] A TSS may receive a MessageInContext message that refers to a context that does not exist at the TSS. This can occur either because the context never existed at the TSS or because it has been discarded by the TSS. In either case, the TSS shall return an exception containing a ContextError service context element with major and minor error codes indicating that the referenced context does not exist. The exception and error values to be returned are defined in Section 16.3.4, “CSS State Machine,” on page 16-25.

[31] The processing of a MessageInContext message that arrives on a one-way call shall be the same as for an ordinary call, except that the TSS will not return a ContextError when the referenced context does not exist.

16.2.3 Authorization Token Format

[32] The **authorization_token** field of the EstablishContext message of the Security Attribute Service context element is used to carry a sequence (0 or more) of typed representations of authorization data. The AuthorizationElementType defines the contents and encoding of the contents of **the_element** field.

```
typedef unsigned long AuthorizationElementType;

struct AuthorizationElement {
   AuthorizationElementType   the_type;
   sequence <octet>           the_element;
};
typedef sequence <CSI::X509AuthorizationElement> AuthorizationToken;

const AuthorizationElementType X509AttributeCertChain = 1;
```

[33] This specification has defined one element encoding type, an X509AttributeCertChain. For this type, the field **the_element** contains an encapsulation octet stream containing an ASN.1 type composed of an X.509 AttributeCertificate and a sequence of 0 or more X.509 Certificates. The corresponding ASN.1 definition appears below:

```
VerifyingCertChain ::= SEQUENCE OF Certificate

AttributeCertChain ::= SEQUENCE {
   attributeCert    AttributeCertificate,
   certificateChain VerifyingCertChain,
}
```

[34] The chain of identity certificates may be provided to certify the attribute certificate. Each certificate in the chain shall directly certify the one preceding it. The first certificate in the chain shall certify the attribute certificate. The ASN.1 representation of Certificate shall be as defined in [IETF RFC 2459]. The ASN.1 representation of AttributeCertificate shall be as defined in [IETF ID PKIXAC].

Extensions of the IETF AC Profile for CSIv2

[35] The **extensions** field of the X.509 Attribute Certificates (AC) provides for the association of additional attributes with the holder or subject of the AC.

[36] Each extension includes an **extnID** (an object identifier), an **extnValue** (an octet string), and a **critical** field (a boolean). The **extnID** identifies the extension, and the **extnValu**e contains the value of the instance of the identified extension. The **critical** field indicates whether a certificate-using system shall reject the certificate if it does not recognize the extension. If the critical field is set to TRUE and the extension is not recognized (by its **extnID**), then the certificate shall be rejected. A non-critical extension that is not recognized may be ignored.

```
Extensions ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE {
   extnID     OBJECT IDENTIFIER,
   critical   BOOLEAN DEFAULT FALSE,
   extnValue   OCTET STRING
}
```

[37] [IETF ID PKIXAC] defines a profile for ACs that defines a collection of extensions that may be used in ACs that conform to the profile. An AC that includes any subset of these extensions conforms to the profile. An AC that includes any other critical extension does not conform to the profile. An AC that includes any other non-critical extension conforms to the profile.

[38] The CSIv2 AC profile adds the Proxy Info extension to the collection of extensions defined by the IETF profile. This critical extension may be used to define who may act as proxy for the AC subject. Refer to [IETF ID PKIXAC] for the details of the format and semantics of the Proxy Info extension.

[39] A TSS shall reject a security context that contains an authorization element of type X509AttributeCertChain that contains critical extensions or attributes not recognized by the TSS. In this case, the TSS shall return a ContextError service context element containing major and minor error codes indicating the evidence is invalid (that is, “Invalid evidence”) as defined in Section 16.3.5, “ContextError Values and Exceptions,” on page 16-28.

16.2.4 Client Authentication Token Format

[40] A CSIv2 client authentication token is a mechanism-specific GSS initial context token. It contains a mechanism type identifier (an object identifier) and the mechanism-specific evidence (that is, the authenticator) required to authenticate the client.

[41] The following ASN.1 basic token definition describes the format of all GSSAPI initial context tokens. The definition of the inner context tokens is mechanism-specific.

```
-- basic Token Format
[APPLICATION 0] IMPLICIT SEQUENCE {
    thisMech MechType
        -- MechType is an Object Identifier
    innerContextToken ANY DEFINED BY thisMech
        -- contents mechanism specific
};
```

[42] The client authentication token has been designed to accommodate the initial context token corresponding to any GSSAPI mechanism. Implementations are free to employ GSSAPI mechanisms other than those required for conformance to CSIv2, such as Kerberos.

[43] The format of the mechanism OID in GSS initial context tokens is defined in [IETF RFC 2743] Section 3.1, "Mechanism-Independent Token Format," pp. 81-82.

Username Password GSS Mechanism (GSSUP)

[44] This specification defines a GSSAPI mechanism to support the delivery of authentication secrets above the transport such that they may be applied by a TSS to authenticate clients at shared secret authentication systems.

[45] The GSSUP mechanism assumes that transport layer security, such as that provided by SSL/TLS, will be used to achieve confidentiality and trust in server, such that the contents of the initial context token do not have to be protected against exposures that occur as the result of networking.

[46] The object identifier allocated for the GSSUP mechanism is defined as follows:

```
{ iso-itu-t (2) international-organization (23) omg (130) security (1)
  authentication (1) gssup-mechanism (1) }
```

GSSUP Initial Context Token

[47] For the GSSUP mechanism, only an inner context token corresponding to the initial context token is defined.

[48] The format of a GSSUP initial context token shall be as defined in [IETF RFC 2743] Section 3.1, "Mechanism-Independent Token Format," pp. 81-82. This GSSToken shall contain an ASN.1 tag followed by a token length, an authentication mechanism identifier, and a CDR encoded sequence of octets corresponding to a GSSUP inner context token as defined by the type GSSUP::InitialContextToken in Section 16.9.6, "Module CSI - Common Secure Interoperability," on page 16-69 (and repeated below).

```
typedef sequence <octet> GSSToken;
sequence <octet> UTF8String;
typedef UTF8String NameValue;

struct ScopedName {
   NameValue name_scope;
   NameValue name_value;
};

// GSSUP::InitialContextToken
struct InitialContextToken {
   Security::ScopedName username;
   Security::UTF8String password;
};
```

[49] The **username** component of a GSSUP::InitialContextToken includes a **name_scope** that identifies an authentication domain or realm and a **name_value** that identifies a user in the domain. The **name_scope** and **name_value** are each represented as a sequence of 0 or more UTF8 characters. If the scope is missing (that is, if it contains a sequence of length 0), the **name_value** is interpreted in the default name scope of the target.

16.2.5 *Identity Token Format*

[50] An identity token is used in an EstablishContext message to carry a "spoken for" or asserted identity. The following table lists the five identity token types and defines the type of identity value that may be carried by each of the token types.

Table 16-2 Identity Token Types

IdentityTokenType (Union Discriminator)	**Meaning**
ITTAbsent	Identity token is absent; the message conveys no representation of identity assertion
ITTAnonymous	Identity token is being used to assert a valueless representation of an unauthenticated caller
ITTPrincipalName	Identity token contains an encapsulation octet stream containing a GSS mechanism-independent exported name object as defined in [IETF RFC 2743]
ITTDistinguishedName	Identity token contains an encapsulation octet stream containing an ASN.1 encoding of an X.501 distinguished name
ITTX509CertChain	Identity token contains an encapsulation octet stream containing an ASN.1 encoding of a chain of X.509 identity certificates

[51] Identity tokens of type ITTX509CertChain contain an ASN.1 encoding of a sequence of 1 or more X.509 certificates. The asserted identity may be extracted as a distinguished name from the subject field of the first certificate. Subsequent certificates shall directly certify the certificate they follow. The ASN.1 encoding of identity tokens of this type is defined as follows:

CertificateChain ::= SEQUENCE SIZE (1..MAX) Certificate

[52] Interpretation of identity tokens that carry a GSS mechanism-independent exported name object (that is, an identity token type of ITTPrincipalName) is dependent on support for GSS mechanism-specific name manipulation functionality.

[53] A target that supports identity assertion shall include in its IORs the complete list of GSS mechanisms for which it supports identity assertions using an identity token of type ITTPrincipalName. A TSS shall reject requests that carry identity tokens of type ITTPrincipalName constructed using a GSS mechanism that is not among the GSS mechanisms supported by the target. The definition of a target's list of supported GSS naming mechanisms is defined in Section , "struct SAS_ContextSec," on page 16-37.

[54] Asserting entities may choose to overcome limitations in a target's supported mechanisms by mapping GSS mechanism-specific identities to distinguished names or certificates. The specifics of such mapping mechanisms are outside the scope of this specification.

16.2.6 Principal Names and Distinguished Names

[55] Principal names are carried in EstablishContext messages of the SAS protocol, where they may appear in the **identity_token** (the ITTPrincipalName discriminated type of an IdentityTokenType) or in the **client_authentication_token**, which is a GSS initial context token.

[56] Principal names are also present in the compound mechanisms defined within a TAG_CSI_SEC_MECH_LIST tagged component within IORs. The **target_name** field of the AS_ContextSec structure may contain a sequence of principal names corresponding to the authentication identities of the target (see Section , "struct AS_ContextSec," on page 16-36). A principal name may be used as one variant of the ServiceSpecificName form used to identify one of the **privilege_authorities** within the SAS_ContextSec structure of a compound mechanism definition within a target IOR (see Section , "struct SAS_ContextSec," on page 16-37).

[57] The principal names appearing in initial context tokens are in mechanism-specific (that is, internal) form, and may be converted to GSS mechanism-independent exported name object format (that is, an external form) by calling a mechanism-specific implementation of GSS_Export_name. The inverse translation is performed by a mechanism-specific implementation of GSS_Import_name. A mechanism-specific implementation of GSS_Display_name allows its caller to convert an internal name representation into a printable form with an associated mechanism type identifier.[7]

7. As defined in [IETF RFC 2743] "Generic Security Service Application Program Interface Version 2, Update 1", J. Linn, January 2000.

[58] The principal names in identity tokens — those in the **target_name** field of AS_ContextSec structures and those in the **privilege_authorities** field of SAS_ContextSec structures — are in external form (GSS_NT_ExportedName), and may be converted to internal form by calling the appropriate mechanism-specific GSS_import_name function.

[59] Distinguished names may appear within an identity token, either as an asserted identity or indirectly as the subject distinguished name within an asserted X.509 Identity Certificate. Distinguished names may also be derived from the underlying transport authentication layer if client authentication is done using SSL certificates. Distinguished names may also be used as a form of GeneralName in the GeneralNames variant of the ServiceSpecificName type. The ServiceSpecificName type is used to identify **privilege_authorities** within the SAS_ContextSec structure of a compound mechanism definition within a target IOR.

16.3 Security Attribute Service Protocol

16.3.1 Compound Mechanisms

[60] The SAS protocol combines common authorization (security attribute) functionality with client authentication functionality and is intended to be used in conjunction with a transport-layer security mechanism, so that there may be as many as three protocol layers of security functionality. This section describes the semantics of the compound security mechanisms that may be realized using this interoperability architecture.

[61] The three protocol layers build on top of each other. The transport layer is at the bottom. The client authentication functionality of the SAS protocol provides a way to layer additional client authentication functionality above the transport layer. The common authorization functionality provides a way to layer security attribute functionality above the authentication layers. Any or all of the layers may be absent.

[62] A target describes in its IORs the CSI compound security mechanisms it supports. Each mechanism defines a combination of layer-specific security functionality supported by the target, as defined in Section , “TAG_CSI_SEC_MECH_LIST,” on page 16-35.

[63] The mechanisms a client uses to interact with a target shall be compatible with the target’s capabilities and sufficient to satisfy its requirements.

Context Validation

[64] A target indicates its requirements for client authentication in its IORs. The layers at which a CSS authenticates to a TSS shall satisfy the requirements established by the target (see the description in Section 16.5.1, “Target Security Configuration,” on page 16-30). When a CSS attempts to authenticate with a TSS using the client authentication functionality of the SAS context layer protocol (by including a client_authentication_token in an EstablishContext message), the authentication

context established in the TSS will reflect the result of the service context authentication (after having satisfied the target's requirement for transport level authentication, if any).

[65] If the service context authentication fails, the following shall happen:

- The request shall be rejected, whether or not authentication is required by the target.
- An exception containing a ContextError service context element shall be returned to the CSS. The ContextError service context element shall contain major and minor status codes indicating that client authentication failed.

[66] If the request does not include a client_authentication_token, the client authentication identity is derived from the transport layer.

[67] When a request includes an identity token, the TSS shall determine if the identity established as the client authentication identity is trusted to assert the identity represented in the identity token.

[68] A TSS that does not support authorization-token-based delegation (see Chapter 9, "Conformance Levels") shall evaluate trust by applying the client authentication identity and the asserted identity to trust rules stored at the target. We call the evaluation of trust based on rules of the target a backward trust evaluation.

[69] When a TSS that supports authorization-token-based delegation receives a request that includes both an identity token and an authorization token with embedded proxy attributes, the TSS shall evaluate trust by determining whether the proxy attributes were established (that is, signed) by a privilege authority acceptable to the target and whether the client authentication identity is included in the identities named in the proxy attributes. We call the evaluation of trust based on rules provided by the caller a forward trust evaluation. A TSS shall not accept requests that failed a forward trust evaluation based on a backward trust evaluation.

[70] A TSS shall determine that a trusted identity established in the authentication layer(s) is trusted to assert exactly the same identity (in terms of identifier value and identification mechanism) in an identity token.

[71] In either case of forward or backward trust evaluation, if trust is established, the context is considered correctly formed. Otherwise, the TSS shall reject the request by returning an exception containing a ContextError service context element. The ContextError element shall contain major and minor status codes indicating that the evidence was invalid.

[72] If a request includes an authorization token but does not include an identity token, the TSS shall ensure that the access identity named in the authorization token is the same as the client authentication identity. If the request includes an identity token, the TSS shall ensure that the access identity is the same as the identity in the identity token. A TSS that supports authorization-token-based privilege attributes shall reject any request that does not satisfy this constraint and return an exception containing a ContextError service context element. The ContextError element shall contain major and minor status codes indicating that the evidence was invalid.

[73] When a request includes an authorization token, it is the responsibility of the TSS to determine if the target trusts the authorities that signed the privileges in the token. A TSS that supports authorization-token-based privilege attributes shall reject any request with an authorization token that contains privilege information signed by an authority that is not trusted by the target. In this case, the TSS shall return an exception containing a ContextError service context element. The ContextError element shall contain major and minor status codes indicating that the evidence was invalid.

Legend for Request Principal Interpretations

[74] This section serves as a key to the invocation scenarios represented in Tables Table 16-3, Table 16-4, and Table 16-5. The three tables describe the interpretation of security context information arriving at a target object from a calling object, object 2, that may have been called by another object, object 1. The authentication identity of object 2, as seen by the target object, may have been established in the transport layer, or the SAS context layer, or both. If the authentication identity was established at the transport layer it is referred to as $P2^A$. If the authentication identity was established at the SAS context layer it is referred to as $P2^B$. The authentication identity seen by object 2 when it is called by another object (that is, object 1) is referred to as P1, the authentication identity of object 1. No distinction is made between the transport and SAS layer authentication identities of object 1 as seen by object 2. Object 1 may also call object 2 anonymously.

[75] P1 is also used to represent a non-anonymous identity that may be asserted by object 2 when it calls the target object. When object 2 calls the target object, it may include an asserted identity in the form of an identity token in its SAS layer context. The asserted identity may be the anonymous identity or, a non-anonymous identity (represented by P1). When object 2 asserts an identity to the target object, it may (or may not) establish proof of its own identity by authenticating at either or both of the transport ($P2^A$), or SAS ($P2^B$) layers. When the target object receives a request made with an asserted identity, the target object will determine if it trusts the client authentication identity (that of object 2, or P2) acting as proxy for the asserted identity (that of object 1, or P1).

[76] When object 2 asserts a non-anonymous identity to the target object, it may include with its request a SAS layer authorization token containing PACs. Each PAC may include an attribute that assigns proxy to a collection of identities that are endorsed by the authority that created the PAC to assert the identity to which the privileges in the PAC apply. When the target object receives a request made with an asserted identity and an authorization token containing proxy rules, the target object will use the proxy rules to determine if it may trust the client authentication identity ($P2^A$ or $P2^B$) as proxy for the asserted identity(P1)

.

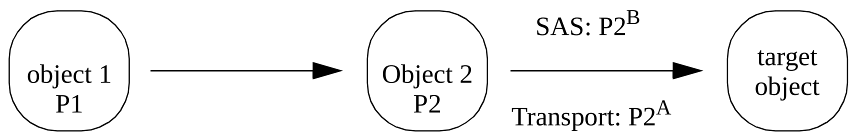


Figure 16-2 Invocation Scenarios

Anonymous Identity Assertion

[77] The anonymous identity is used to represent an unathenticated entity. To assert an anonymous caller identity, a CSS (perhaps acting as an intermediate) shall include a SAS context element containing an EstablishContext message with an identity_token containing the anonymous IdentityTokenType in its request.

Presumed Trust

[78] Presumed trust is a special case of the evaluation of identity assertions by a TSS. In presumed trust, a TSS accepts identity assertions based on the fact of their occurrence and without consideration of the authentication identity of the asserting entity. The presumption is that communications are constrained such that only trusted entities are capable of asserting an identity to the TSS.

Failed Trust Evaluations

[79] Table 16-3 shows the circumstances under which the interpretation of caller credentials by a TSS results in a failed trust evaluation. None of these circumstances correspond to presumed trust, where trust evaluations are not performed (and therefore cannot fail).

Table 16-3 Conditions under which Trust Evaluation Fails

SAS Identity Token Identity	**Transport Client Principal**	**SAS Client Authentication Principal**	**Does Target Trust P2, or Is P2 Named as Proxy in Authorization Elements?**
P1	None	None	Not Applicable
P1	None	$P2^B$	No (with respect to $P2^{B)}$
P1	$P2^A$	None	No (with respect to $P2^{A)}$
P1	$P2^A$	$P2^B$	No (with respect to $P2^B$)

[80] A failed trust evaluation shall result in the request being rejected with an indication that client authentication failed.

Request Principal Interpretations

[81] The entries in Table 16-4 describe the interpretation of client credentials by a TSS after an incoming call has satisfied the target's security requirements and has been validated by the TSS..

Table 16-4 TSS Interpretation of Client Credentials After Validation

SAS Identity Token Identity	Transport Client Principal	SAS Client Authentication Principal	Client Principal is Trusted	Invocation Principal	Scenario
Absent	None	None	Not applicable	Anonymous	Unauthenticated
Absent	None	$P2^B$	Not applicable	P2	Client authentication
Absent	$P2^A$	None	Not applicable	P2	Client authentication
Absent	$P2^A$	$P2^B$ (by rule 1[1])	Not applicable	$P2^B$	Client authentication
P1	None	None	Yes if rule 2[2]	P1	identity assertion
P1	None	$P2^B$	Yes if rule 2 or rule 3[3]	P1	identity assertion
P1	$P2^A$	None	Yes if rule 2 or rule 3	P1	identity assertion
P1	$P2^A$	$P2^B$ (by rule 1)	Yes if rule 2 or rule 3	P1	identity assertion
Anonymous	None	None	Yes if rule 4[4]	Anonymous	assertion of anonymous
Anonymous	None	$P2^B$	Yes if rule 4	Anonymous	assertion of anonymous
Anonymous	$P2^A$	None	Yes if rule 4	Anonymous	assertion of anonymous
Anonymous	$P2^A$	$P2^B$ (by rule 1)	Yes if rule 4	Anonymous	assertion of anonymous

1. Rule 1: TSS trusts $P2^A$ to use authenticator for $P2^B$ is implied by $P2^B$ having been authenticated.
2. Rule 2: TSS presumes trust in transport to accept None, $P2^A$, or $P2^B$ speaking for P1.
3. Rule 3: TSS trusts $P2^A$, or $P2^B$ to speak for P1.
4. Rule 4: TSS trusts None, $P2^A$, or $P2^B$ to speak for Anonymous. A TSS shall support the configuration of rule 4, such that Anonymous identity assertions are accepted independent of authentication of the asserter.

[82] The entries in Table 16-5 describe additional TSS interpretation rules to support delegation. These rules have been separated from those in Table 16-4, because they describe functionality required of implementations that conform to a higher level of secure interoperability as defined in Section 16.6.3, "Conformance Level 2," on page 16-44. The entries in Table 16-5 correspond to invocations that carry an identity token and an authorization token with embedded delegation token (that is, a proxy endorsement attribute) in an EstablishContext service context element. Invocations that do not carry all of these tokens are represented in Table 16-4.

[83] An authorization token may contain authorization elements that contain proxy statements, which endorse principals to proxy for other entities. Table 16-5 describes delegation scenarios in which endorsements from the issuer of the authorization element authorize the authenticated identity, which is $P2^A$ or $P2^B$, to proxy for the asserted identity. In this table, the column “Proxies Named in Authorization Element” defines the identities who are endorsed by the authorization element to proxy for P1, the asserted identity and the subject of the authorization element. The value “Any” indicates that the authorization element contains a blanket endorsement, such that as far as its issuer is concerned, any identity may proxy for P1. The outcomes described in Table 16-5 assume that the TSS trusts the issuer of the authorization element to endorse principals to proxy for others..

Table 16-5 Additional TSS Rules to Support Delegation

SAS Identity Token Identity	**Transport Client Principal**	**SAS Client Authentication Principal**	**Proxies Named in Authorization Element**	**Invocation Principal**	**Scenario**
P1	None	$P2^B$	Any	P1	Delegation
P1	$P2^A$	None	Any	P1	Delegation
P1	$P2^A$	$P2^B$	Any	P1	Delegation
P1	None	$P2^B$	Restricted to set including $P2^B$	P1	Restricted delegation
P1	$P2^A$	None	Restricted to set including $P2^A$	P1	Restricted delegation
P1	$P2^A$	$P2^B$	Restricted to set including $P2^B$	P1	Restricted delegation

16.3.2 Session Semantics

[84] This section describes the negotiation of security contexts between a CSS and a TSS. A TSS is said to be stateless if it does not operate in the mode of accepting reusable (that is, stateful) security contexts. A TSS that accepts reusable security contexts is said to be stateful. A CSS is said to be stateless if it operates in the mode of establishing transient, non-reusable (that is, stateless) security contexts. A CSS that issues requests to establish reusable security contexts is said to be stateful.

Negotiation of Statefulness

[85] A client initiates a stateless interaction by specifying a client_context_id of 0. A client issues a request to establish a stateful context by including a nonzero client_context_id in an EstablishContext message.

[86] When a stateless TSS receives a request to establish a stateful session, the TSS shall attempt to validate the security tokens bound to the request. If the validation fails, an exception containing an appropriate ContextError service context element shall be returned to the client. If the validation succeeds, the TSS shall negotiate to stateless by responding with a CompleteEstablishContext message with context_stateful set to false.

[87] A client that initiates a stateful interaction shall be capable of accepting that the target negotiated the context to stateless.

Stateful/Reusable Contexts

[88] Each transport layer session defines a context identifier number *scope*. The CSS selects context identifiers for use within a scope.

[89] A CSS may use the EstablishContext message to issue multiple concurrent requests to establish a stateful security context within a scope.

[90] To avoid duplicate sessions, when the stateful EstablishContext requests sent within a scope carry equivalent security contexts, the CSS shall assign to them the same nonzero client_context_id.

[91] Within a scope, a TSS shall reject any request to establish a stateful context that carries a different security context from an established context with the same client_context_id. In this case, an exception containing a ContextError service context element shall be returned to the caller.

[92] Two security contexts are equivalent if all of the authentication, identity, and authorization tokens match both in existence and in value. Token values shall be evaluated for equivalence by comparing the corresponding byte sequences used to carry the tokens in EstablishContext messages.

[93] When a target that supports stateful contexts receives a request to establish a stateful context, the TSS shall attempt to validate the security tokens in the EstablishContext element. If the validation succeeds, the request shall be accepted, and the reply (if there is one) shall carry a CompleteEstablishContext element that indicates (that is, **context_stateful** = true) that the context is available at the TSS for the caller's reuse. If the validation fails, an exception containing an appropriate ContextError service context element shall be returned to the caller.

[94] A TSS that accepts stateful contexts shall bear the responsibility for managing the lifecycle of these sessions. Clients that reuse stateful contexts shall capable of processing replies that indicate that an established stateful context has been unilaterally discarded by the TSS.

[95] A TSS shall not establish a stateful context in response to a request to establish a stateless context (that is, one with a **client_context_id** of zero)

[96] A TSS that supports stateful contexts may negotiate a request to establish a stateful context to a stateless context in order to preserve resources. It may do so only if it does not already have an established matching stateful context.

[97] Conversely, a stateful TSS that has negotiated a request to stateless may respond statefully to a subsequent context with the same (non-zero) **client_context_id**.

Relationship to Transport-Layer

[98] A SAS context shall not persist beyond the lifetime of the transport-layer secure association over which it was established.

[99] Stateful SAS contexts are not compatible with transports that do not make the relationship between the connection and the association transparent.

16.3.3 TSS State Machine

[100] The TSS state machine is defined in the state diagram, Figure 16-2 on page 16-23 and in the TSS state table, Table 16-6 on page 16-24. Each TSS call thread shall operate independently with respect to this state machine. Where necessary, thread synchronization at shared state shall be handled in the actions called by this state machine.

[101] An ORB must not invoke the TSS state machine if the target object does not exist at the ORB. The TSS state machine has no capacity to reject or forward[8] a request because the target object does not exist, and must rely on the ORB to only invoke the TSS when the target object exists at the ORB.

[102] In response to a one-way call, a TSS shall not perform any of the send actions described by the state machine.

[103] The shaded rows in Table 16-6 indicate transitions and states that do not exist in a stateless implementation of the SAS protocol.

[104] The state names, function names, and function signatures that appear in the state diagram and the state table are not prescriptive.

8. A TSS uses the LOCATION_FORWARD status to return an IOR containing up-to-date security mechanism configuration for an existing object.

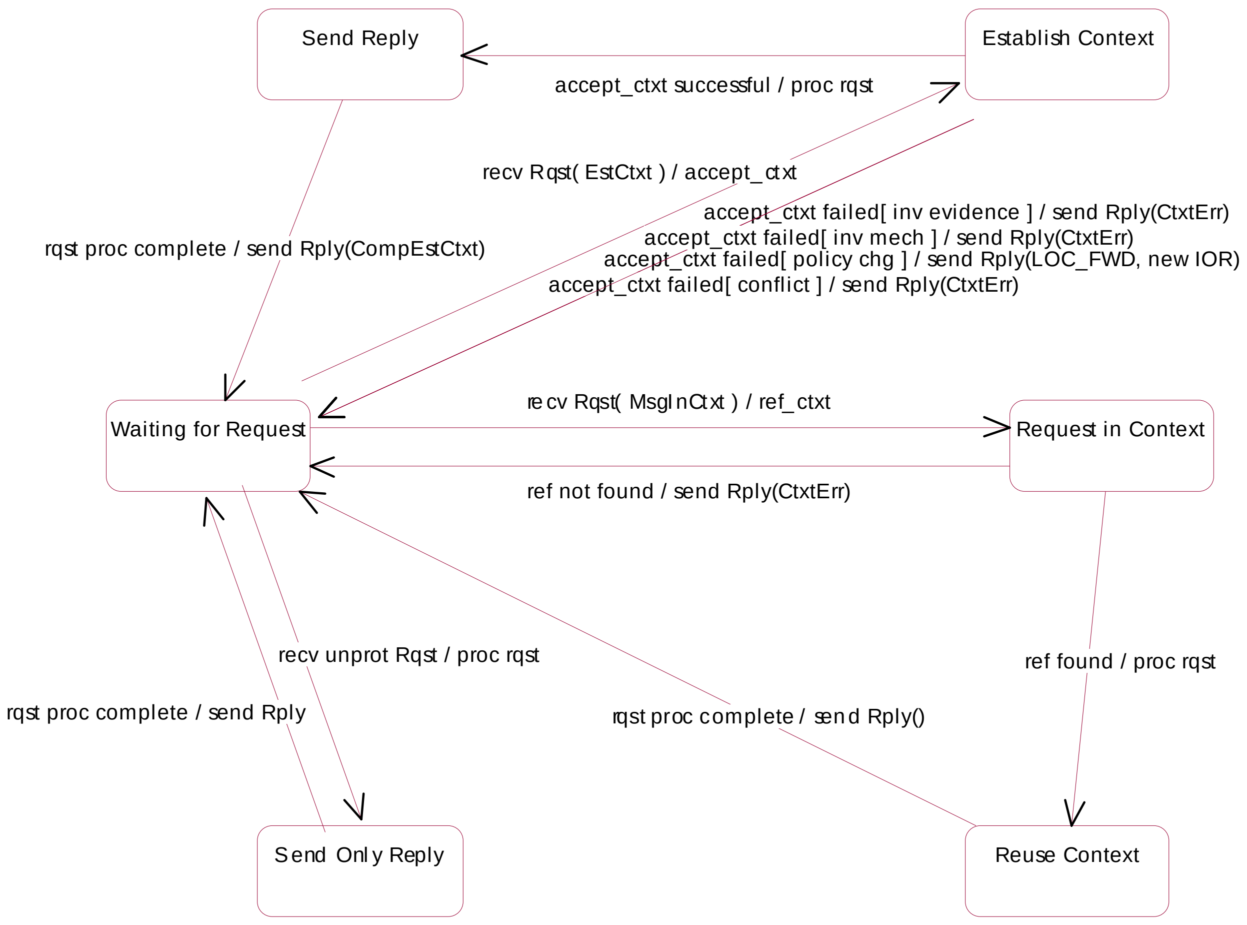

Send Reply
Establish Context
accept_ctxt successful / proc rqst
recv Rqst(EstCtxt) / accept_ctxt
accept_ctxt failed[inv evidence] / send Rply(CtxtErr)
accept_ctxt failed[inv mech] / send Rply(CtxtErr)
accept_ctxt failed[policy chg] / send Rply(LOC_FWD, new IOR)
accept_ctxt failed[conflict] / send Rply(CtxtErr)
rqst proc complete / send Rply(CompEstCtxt)
Waiting for Request
recv Rqst(MsgInCtxt) / ref_ctxt
Request in Context
ref not found / send Rply(CtxtErr)
recv unprot Rqst / proc rqst
ref found / proc rqst
rqst proc complete / send Rply
rqst proc complete / send Rply()
Send Only Reply
Reuse Context

Figure 16-2 TSS State Machine

Table 16-6 TSS State Table

	State	Event	Action	New State
1	Waiting for Request	receive unprotected Request	process request	Send Only Reply
		receive Request + EstablishContext {client_context_id = N, tokens}	accept_context(tokens, N, Out stateful)	Establish Context
		receive Request + MessageInContext {client_context_id = N, discard_context = D}	reference_context(N)	Request In Context
2	Send Only Reply	request processing completed	send Reply	Waiting for Request
3	Send Reply	request processing completed	send Reply + CompleteEstablishContext { N, stateful}	Waiting For Request
4	Establish Context	accept_context (tokens, N, Out stateful) returned success	process request	Send Reply
		accept_context (tokens, N, Out stateful) returned failure (invalid evidence)	send exception + ContextError (invalid evidence)	Waiting for Request
		accept_context (tokens, N, Out stateful) returned failure (invalid mechanism)	send exception + ContextError (invalid mechanism)	Waiting for Request
		accept_context (tokens, N, Out stateful) returned failure (policy change)	send Reply + LOCATION_FORWARD status + updated IOR	Waiting for Request
		accept_context (tokens, N, Out stateful) returned failure (conflicting evidence)	send exception + ContextError (conflicting evidence)	Waiting for Request
5	Request in Context	reference_context(N) returned reference	process request	Reuse Context
		reference_context(N) returned empty reference	send exception + ContextError (context does not exist)	Waiting for Request
6	Reuse Context	request processing completed	send Reply if (D) discard_context(N)	Waiting for Request

TSS State Machine Actions

[105] This section defines the intended semantics of the actions appearing in the TSS state machine. As noted above, the function names and function signatures are not prescriptive.

- accept_context (tokens, N, Out stateful)

  This action validates the security context captured in the tokens including ensuring that they are compatible with the mechanisms supported by the target object. If a context is not validated, **accept_context** returns error codes that describe the reason the context was rejected.

  When called by a stateless TSS, **accept_context** always returns false in the output argument "**stateful**".

When called by a stateful TSS, **accept_context** may (depending on the effective policy of the target object) attempt to record state corresponding to the context. If state for the identified context already exists and the received tokens are not equivalent to those captured in the existing context, **accept_context** shall reject the context. If the context state either already existed, or was recorded, **accept_context** returns true in the output argument "**stateful**".

An implementation of accept_context shall implement the error semantics defined in the following table.

Table 16-7 Accept Context Error Semantics

Semantic	Returned Error Code
tokens match mechanism definition of target object but could not be validated	Invalid evidence
context has non-zero client_context_id that matches that of an exiting context but tokens are not equivalent to those used to establish the existing context	Conflicting evidence
the mechanism configuration of the target object has changed and request indicates that CSS is not aware of the current mechanism configuration	Policy change
the mechanism configuration of the target object has not changed, and request is not consistent with target mechanism configuration	Invalid mechanism

When **accept_context** returns any of **Invalid evidence, Conflicting evidence,** or **Invalid mechanism**, the TSS shall reject the request and send a NO_PERMISSION exception containing a ContextError service context element with error codes as defined in Table 16-9 on page 16-29. When accept_context returns **Policy change**, the TSS action shall reject the request and return a reply with status LOCATION_FORWARD and containing a new IOR for the target object that contains an up-to-date representation of the target's security mechanism configuration.

- reference_context (N)

  If there is an existing context with **client_context_id** = **N**, reference_context returns a reference to it. Otherwise, **reference_context** returns an empty reference.

- discard_context (N)

  If context **N** exists and it is not needed to complete the processing of another thread, **discard_context** causes the context to be deleted.

16.3.4 CSS State Machine

[106] A proposed implementation of the CSS state machine is defined in the state diagram, Figure 16-3 on page 16-26, and in the CSS state table, Table 16-8 on page 16-27. Each CSS call thread shall operate independently with respect to this state machine. Where necessary, thread synchronization at shared state shall be handled in the actions called by this state machine.

[107] When a CSS processes a one-way call, it returns to the caller and sets its next state to done, as no response will be sent by the TSS.

[108] The shaded rows in the state table indicate transitions and states that need not exist in a stateless CSS client side implementation.

[109] The state names, function names, and function signatures that appear in the state diagram and state table are not prescriptive.

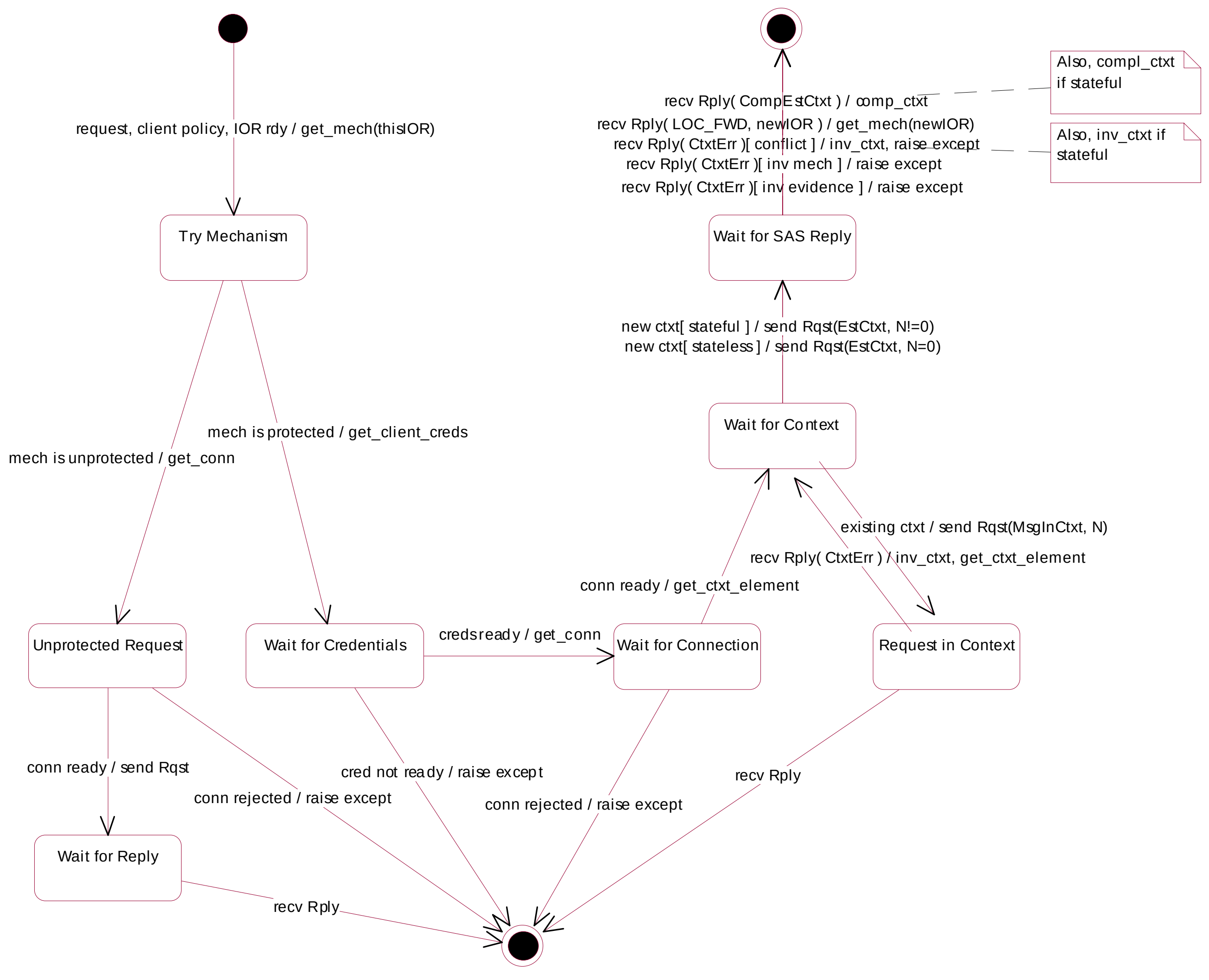

request, client policy, IOR rdy / get_mech(thisIOR)
Try Mechanism
mech is unprotected / get_conn
mech is protected / get_client_creds
Unprotected Request
Wait for Credentials
creds ready / get_conn
Wait for Connection
Request in Context
conn ready / send Rqst
conn rejected / raise except
cred not ready / raise except
conn rejected / raise except
recv Rply
Wait for Reply
recv Rply
conn ready / get_ctxt_element
Wait for Context
existing ctxt / send Rqst(MsgInCtxt, N)
recv Rply(CtxtErr) / inv_ctxt, get_ctxt_element
new ctxt[stateful] / send Rqst(EstCtxt, N!=0)
new ctxt[stateless] / send Rqst(EstCtxt, N=0)
Wait for SAS Reply
recv Rply(CompEstCtxt) / comp_ctxt
recv Rply(LOC_FWD, newIOR) / get_mech(newIOR)
recv Rply(CtxtErr)[conflict] / inv_ctxt, raise except
recv Rply(CtxtErr)[inv mech] / raise except
recv Rply(CtxtErr)[inv evidence] / raise except
Also, compl_ctxt if stateful
Also, inv_ctxt if stateful

Figure 16-3 CSS State Machine

Table 16-8 CSS State Table

	State	Event	Action	New State
1	start	Request + client policy + IOR ready to send	get_mechanism (policy, thisIOR, Out mech)	Try Mechanism
2	Try Mechanism	the selected mechanism is unprotected	get_connection (mech, Out c)	Unprotected Request
		the selected mechanism is protected	get_client_creds (policy, mech, Out creds)	Wait for Credentials
3	Unprotected Request	connection ready	send request	Wait for Reply
		connection rejected	raise exception and return to caller[1]	done
4	Wait for Reply	receive reply	return to caller	done
5	Wait for Credentials	client credentials ready	get_connection (policy, mech, creds, Out c)	Wait for Connection
		necessary credentials not obtained	raise exception and return to caller[2]	done
6	Wait for Connection	connection ready	get_context_element (c, policy, creds, mech, Out element)	Wait for Context
		connection rejected	raise exception and return to caller[3]	done
7	Wait for Context	get_context_element returned EstablishContext {N = 0, tokens}	send Request + EstablishContext {client_context_id = N = 0, tokens}	Wait for SAS Reply
		get_context_element returned EstablishContext {N != 0, tokens}	send Request + EstablishContext {client_context_id = N != 0, tokens}	Wait for SAS Reply
		get_context_element returned MessageInContext {N != 0, D}	send Request + MessageInContext {client_context_id = N != 0, D}	Request In Context
8	Wait for SAS Reply	receive exception + ContextError (invalid evidence)	raise exception and return to caller[4]	done
		receive exception + ContextError (invalid mechanism)	raise exception and return to caller	done
		receive exception + ContextError (conflicting evidence)	invalidate_context (c, N)	done
			raise exception and return to caller	
		receive Reply + LOCATION_FORWARD status + updated IOR	return to caller	done
		receive Reply + CompleteEstablishContext {N, context_stateful}	complete_context (c, N, context_stateful)	done
			return to caller	
9	Request in Context	receive exception + ContextError (context does not exist)	invalidate_context (c, N) get_context_element (c, policy, creds, mech, Out element)	Wait for Context
		receive Reply	return to caller	done

1. A CSS may do next mechanism processing, in which case it might call get_next_mechanism(policy,thisIOR) and transition to state Try Mechanism.

2. Same note as 1.

3. Same note as 1.

4. A CSS may re-collect authentication evidence and try again, in which case it might call get_client_creds(policy, mech, Out creds) and transition to state Wait for Credentials.

CSS State Machine Actions

[110] This section defines the intended semantics of the actions appearing in the CSS state machine. As noted above the function names and function signatures are not prescriptive. The descriptions appearing in the following sections are provided to facilitate understanding of the proposed implementation of the CSS state machine.

- get_mechanism (policy, IOR, Out mech)

  Select from the **IOR** a **mechanism** definition that satisfies the client policy.
- get_client_creds (policy, mech, Out creds)

  Get the client **credentials** as necessary to satisfy the client **policy** and the target policy in the **mechanism**.
- get_connection (mech, Out c)

  Open a connection based on the port information in the **mechanism** argument.
- get_connection (policy, mech, creds, Out c)

  Open a secure connection based on the client **policy**, the target policy in the **mechanism** argument, and using the client credentials in the **creds** argument.
- get_context_element(c, policy, creds, mech, Out element)

  In the scope of connection **c**, use the client **creds** to create a SAS protocol context element that satisfies the client **policy** and the target policy in the **mechanism**. If the CSS supports reusable contexts, and the client policy is to establish a reusable context, the CSS allocates a **client_context_id**, and initializes a context element in the context table of the connection.
- invalidate_context (c, N)

  Mark context **N** in connection scope **c** as invalid such that no more requests may (re)use it.
- complete_context (c, N, context_stateful)

  This action applies the contents of a returned CompleteEstablishContext message to context **N**, in connection scope **c,** to change its state to completed. In a stateful CSS, get_context_element will not return a MessageInContext element until complete_context is called with context_stateful true.

16.3.5 ContextError Values and Exceptions

[111] Table 16-9 defines the circumstances under which error values and exceptions shall be returned by a TSS. The state and event columns contain states and events appearing in Table 16-6 on page 16-24.

Table 16-9 ContextError Codes and Exceptions

State	Event	Semantic	Major	Minor	Exception
Establish Context	accept_context returned failure	Invalid evidence	1	1	NO_PERMISSION
		Invalid mechanism	1	2	NO_PERMISSION
		Conflicting evidence	1	3	NO_PERMISSION
Request In Context	reference_context (N) returned false	No Context	1	4	NO_PERMISSION

16.4 Transport Security Mechanisms

16.4.1 Transport Layer Interoperability

[112] The secure interoperability architecture that is defined by this specification partitions secure interoperability into three layers: the transport layer, authentication above the transport layer, and the secure attribute layer. This specification defines secure interoperability that uses transport-layer security for message protection and authentication of the target to the client.

16.4.2 Transport Mechanism Configuration

[113] The configuration of transport-layer security mechanisms is specified in IORs. Support for CSI is indicated within an IOR profile by the presence of at most one TAG_CSI_SEC_MECH_LIST tagged component that defines the mechanism configuration pertaining to the profile. This component contains a list of one or more CompoundSecMech structures, each of which defines the layer-specific security mechanisms that comprise a compound mechanism that is supported by the target. This specification does not define support for CSI mechanisms in multiple-component IOR profiles.

[114] Each CompoundSecMech structure contains a **transport_mech** field that defines the transport-layer security mechanism of the compound mechanism. A compound mechanism that does not implement security functionality at the transport layer shall contain the TAG_NULL_TAG component in its **transport_mech** field. Otherwise, the **transport_mech** field shall contain a tagged component that defines a transport protocol and its configuration. Section , “TAG_SSL_SEC_TRANS,” on page 16-33 and Section , “TAG_SECIOP_SEC_TRANS,” on page 16-34 define valid transport-layer components that can be used in the **transport_mech** field.

Recommended SSL/TLS Ciphersuites

[115] This specification recommends that implementations support the following ciphersuites in addition to the mandatory ciphersuites identified in [IETF RFC 2246]. Of these additional ciphersuites, those which use weak encryption keys are only recommended for use in environments where strong encryption of SAS protocol elements (including GSSUP authenticators) and request arguments is not required. Some of the recommended ciphersuites are known to be encumbered by licensing constraints.

- TLS_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_MD5
- TLS_DHE_DSS_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

16.5 Interoperable Object References

16.5.1 Target Security Configuration

[116] A target that supports unprotected IIOP invocations shall specify in the corresponding TAG_INTERNET_IOP profile a nonzero port number at which the target will accept unprotected invocations.[9] A target that supports only protected IIOP invocations shall specify a port number of 0 (zero) in the corresponding TAG_INTERNET_IOP profile. A target may support both protected and unprotected IIOP invocations at the same port, but it is not required to do so.

```
struct IOR {
    string type_id;
    sequence <TaggedProfile> profiles = {
        ProfileId tag = TAG_INTERNET_IOP;
        struct ProfileBody_1_1 profile_data = {
            Version iiop_version;
            string host;
            unsigned short port;
            sequence <octet> object_key;
            sequence <IOP::TaggedComponent> components;
         };
    };
};
```

[117] A target that supports protected invocations shall describe in a CompoundSecMech structure the characteristics of each of the alternative compound security mechanisms that it supports. The CompoundSecMech structure shall be included in a list of such structures in the body of a TAG_CSI_SEC_MECH_LIST tagged component.

9. The OMG has registered port numbers for IIOP (683) and IIOP/SSL (684) with IANA. Although the existence of these reservations does not prescribe their use, it may be useful to recognize these port numbers as defaults for the corresponding protocols.

```
sequence <IOP::TaggedComponent> components = {
    IOP::TaggedComponent {
        ComponentId tag = TAG_CSI_SEC_MECH_LIST;
        sequence <octet> component_data = {
            sequence <IIOP::CompoundSecMech> CompoundSecMechList = {
                CompoundSecMech;
            };
        };
    };
};
```

[118] The order of occurrence of the alternative compound mechanism definitions in a TAG_CSI_SEC_MECH_LIST component indicates the target's mechanism preference. The target prefers mechanism definitions occurring earlier in the list. An IOR profile shall contain at most one TAG_CSI_SEC_MECH_LIST tagged component. An IOR profile that contains multiple TAG_CSI_SEC_MECH_LIST tagged components is malformed and should be rejected by a client implementation.

AssociationOptions Type

[119] The AssociationOptions type is an unsigned short bit mask containing the logical OR of the configured options. The properties of security mechanisms are defined in an IOR in terms of the association options supported and required by the target. A CSS shall be able to interpret the association options defined in Table 16-10.

*Table 16-10*Association Options

Association Option	**target_supports**	**target_requires**
Integrity	Target supports integrity protected messages	Target requires integrity protected messages
Confidentiality	Target supports privacy protected messages	Target requires privacy protected messages
EstablishTrustInTarget	Target can authenticate to a client	Not applicable. This bit should never be set, and should be ignored by CSS
EstablishTrustInClient	Target can authenticate a client	Target requires client authentication
IdentityAssertion	Target accepts asserted caller identities based on trust in the authentication identity of the asserting entity. Target can evaluate trust based on trust rules of the target. If DelegationByClient is set, target can also evaluate trust when provided with a delegation token (that is, a proxy attribute contained in an authorization token).[1]	Not applicable. This bit should never be set, and should be ignored by CSS
DelegationByClient	When it occurs in conjunction with support for IdentityAssertion, this bit indicates that target can evaluate trust in an asserting entity based on a delegation token.[2]	Target requires that CSS provide a delegation token that endorses the target as proxy for the client.[3]

1. A target policy that accepts only identity assertions based on forward trust cannot be communicated in an IOR (although it can be enforced).
2. If an incoming request includes an identity token and a delegation token, the request shall be rejected if the delegation token does not endorse the asserting entity (see Section , “Context Validation,” on page 16-15)
3. A target with DelegationByClient set in **target_requires** shall also have this bit set in **target_supports**. As noted in the table, this has an impact on the target’ s identity assertion policy (if any).

[120] The representation of supported options is used by a client to determine if a mechanism is capable of supporting the client’s security requirements. The supported association options shall be a superset of those required by the target.

[121] When the IdentityAssertion bit is set in **target_support**s, it indicates that the target accepts asserted caller identities based on trust in the authentication identity of the asserting entity. When the DelegationByClient bit is not set, the target will evaluate trust based on rules of the target (that is, a backward trust evaluation). When the IdentityAssertion and DelegationByClient bits are set, they indicate that the target is also capable of evaluating trust in an asserting entity based on trust rules delivered in an authorization token (that is, a forward trust evaluation). A target that can perform a forward trust evaluation does so when trust rules are delivered in an authorization token. Otherwise a backward trust evaluation is performed.

[122] When the DelegationByClient bit is set in **target_requires**, it indicates that the target requires a delegation token to complete the processing of a request. Such circumstances will occur when a target, acting as an intermediate, attempts to issue a request as its caller and sanctioned by the delegation token delivered by its caller.

[123] The rules for interpreting asserted identities in the presence or absence of a delegation token (that is, a proxy attribute contained in an authorization token) are as defined in Section , "Context Validation," on page 16-15.

[124] The security mechanism configuration in an IOR being used by a CSS may (as the result of target policy administration) no longer represent the actual security mechanism configuration of the target object.

Alternative Transport Association Options

[125] Implementations that choose to employ the service context protocol defined in this specification to achieve interoperability over an alternative secure transport (one other than SSL/TLS) may also be required to support the message protection options defined in Table 16-11.

*Table 16-11*Alternative Transport Association Options

Association Option	target_supports	target_requires
DetectReplay	Target can detect replay of requests (and request fragments)	Target requires security associations to detect replay
DetectMisordering	Target can detect sequence errors of request (and request fragments)	Target requires security associations to detect message sequence errors

TAG_SSL_SEC_TRANS

[126] An instance of the TAG_SSL_SEC_TRANS component may occur in the **transport_mech** field within a CompoundSecMech structure in a TAG_CSI_SEC_MECH_LIST component. The semantics associated with existing uses of this component outside of a containing TAG_CSI_SEC_MECH_LIST component are not modified or described by this specification.

[127] When an instance of the TAG_SSL_SEC_TRANS component occurs in a containing TAG_CSI_SEC_MECH_LIST component, it defines the port number at which the target will be listening for SSL/TLS protected invocations. The supported (**target_supports**) and required (**target_requires**) association options defined in the component define the transport level security characteristics of the target (at the port).

```
const IOP::ComponentId TAG_SSL_SEC_TRANS = 20;
struct SSL {
   Security::AssociationOptions target_supports;
   Security::AssociationOptions target_requires;
   unsigned short port;
};
```

[128] Table 16-12, Table 16-13, Table 16-14, and Table 16-15 describe the association option semantics relating to the TAG_SSL_SEC_TRANS tagged component that shall be interpreted by a CSS and enforced by a TSS. The IdentityAssertion and

DelegationByClient association options shall not occur in an instance of this component.

*Table 16-12*Integrity Semantics

Integrity	Semantic
Not supported	None of the ciphersuites supported by the target designate a MAC algorithm
Supported	Target supports one or more ciphersuites that designate a MAC algorithm
Required	All the ciphersuites supported by the target designate a MAC algorithm

*Table 16-13*Confidentiality Semantics

Confidentiality	Semantic
Not supported	None of the ciphersuites supported by the target designate a bulk encryption algorithm[1]
Supported	Target supports one or more ciphersuites that designate a bulk encryption algorithm
Required	All the ciphersuites supported by the target designate a bulk encryption algorithm

1. Bulk encryption algorithms include both block and stream ciphers.

*Table 16-14*EstablishTrustInTarget Semantics

EstablishTrustInTarget	Semantic
Not supported	None of the ciphersuites supported by the target designate a key exchange algorithm that will authenticate the target to the client
Supported	Target supports one or more ciphersuites that designate a key exchange algorithm that will authenticate the target to the client
Required	Not applicable. This bit should never be set, and should be ignored by CSS

*Table 16-15*EstablishTrustInClient Semantics

EstablishTrustInClient	Semantic
Not supported	Target does not support client authentication during the handshake. Moreover, target provides no opportunity for client to authenticate in the handshake (that is, target does not send certficate request message).
Supported	Target provides client with an opportunity to authenticate in handshake. Target will accept connection if client does not authenticate.
Required	Target accepts connections only from clients who successfully authenticate in the handshake

TAG_SECIOP_SEC_TRANS

[129] A tagged component with the TAG_SECIOP_SEC_TRANS tag is a valid component for the **transport_mech** field of the CompoundSecMech structure. The presence of this component indicates the generic use of the SECIOP protocol as a secure transport underneath the CSI mechanisms. A component tagged with this value shall contain the CDR encoding of the SECIOP_SEC_TRANS structure.

[130] The SECIOP_SEC_TRANS structure defines the TCP/IP port for SECIOP messages, the association options pertaining to the particular GSS mechanism being supported, the GSS mechanism identifier, and the target's GSS exported name.

```
const IOP::ComponentId TAG_SECIOP_SEC_TRANS = 35;

struct SECIOP_SEC_TRANS {
   Security::AssociationOptions target_supports;
   Security::AssociationOptions target_requires;
   Security::OID mech_oid;
   Security::GSS_NT_ExportedName target_name;
   unsigned short port;
};
```

[131] Table 16-12, Table 16-13, Table 16-14, and Table 16-15 also describe the association option semantics relating to the TAG_SECIOP_SEC_TRANS tagged component that shall be interpreted by a CSS and enforced by a TSS.

TAG_CSI_SEC_MECH_LIST

[132] This new tagged component, TAG_CSI_SEC_MECH_LIST, is used to describe support in the target for a sequence of one or more compound security mechanisms represented in the **mechanism_list** field of a CompoundSecMechList structure. The mechanism descriptions in the **mechanism_list** occur in decreasing order of target preference.

```
const IOP::ComponentId TAG_CSI_SEC_MECH_LIST = 33;
struct CompoundSecMech {
   Security::AssociationOptions target_requires;
   IOP::TaggedComponent transport_mech;
   CSIIOP::AS_ContextSec as_context_mech;
   CSIIOP::SAS_ContextSec sas_context_mech;
};

struct CompoundSecMechList {
   boolean stateful;
   sequence <CompoundSecMech> mechanism_list;
};
```

[133] The CompoundSecMech structure is used to describe support in the target for a compound security mechanism that may include security functionality that is realized in the transport and/or security functionality realized above the transport in service context. Where a compound security mechanism implements security functionality in the transport layer, the transport functionality shall be represented in a transport-specific component (for example, TAG_SSL_SEC_TRANS) contained in the **transport_mech** field of the CompoundSecMech structure. Where a compound security mechanism implements client authentication functionality in service context, the mechanism shall be represented in an AS_ContextSec structure contained in the **as_context_mech** field of the CompoundSecMech structure. Where a compound security mechanism supports identity assertion or supports authorization attributes

delivered in service context, the mechanism shall be represented in a SAS_ContextSec structure contained in the **sas_context_mech** field of the CompoundSecMech structure.

[134] At least one of the **transport_mech**, **as_context_mech**, or **sas_context_mech** fields shall be configured. The TAG_NULL_TAG component shall be used in the **transport_mech** field to indicate that a mechanism does not implement security functionality at the transport layer. A value of "no bits set" in the **target_supports** field of either the **as_context_mech** or **sas_context_mech** fields shall be used to indicate that the mechanism does not implement security functionality at the corresponding layer.

[135] The **target_requires** field of the CompoundSecMech structure is used to designate a required outcome that shall be satisfied by one or more supporting (but not requiring) layers. The **target_requires** field also represents all the options required independently by the various layers as defined within the mechanism.

[136] Each compound mechanism defines a combination of layer-specific functionality that is supported by the target. A target's mechanism configuration is the sum of the combinations defined in the individual mechanisms.

[137] A value of TRUE in the **stateful** field of the CompoundSecMechList structure indicates that the target supports the establishment of stateful or reusable SAS contexts. This field is provided to assist clients in their selection of a target that supports stateful contexts. It is also provided to sustain implementations that serialize stateful context establishment on the client side as a means to conserve precious server-side authentication capacity.[10]

struct AS_ContextSec

[138] The AS_ContextSec structure is used in the **as_context_mech** field within a CompoundSecMech structure in a TAG_CSI_SEC_MECH_LIST component to describe the client authentication functionality that the target expects to be layered above the transport in service context by means of the **client_authentication_token** of the EstablishContext element of the SAS protocol.

```
struct AS_ContextSec{
   Security::OID client_authentication_mech;
   sequence <Security::GSS_NT_ExportedName> target_names;
   Security::AssociationOptions target_supports;
   Security::AssociationOptions target_requires;
};
```

[139] A value of "no bits set" in the **target_supports** field indicates that the mechanism does not implement client authentication functionality above the transport in service context. In this case, the values present in any of the other fields in this structure are irrelevant.

10. This serialization is only done when an attempt is being made to establish a stateful context.

[140] If the **target_supports** field indicates that the mechanism supports client authentication in service context, then the **client_authentication_mech** field shall contain a GSS OID that identifies the GSS mechanism that the compound mechanism supports for client authentication above the transport.

[141] The target uses the **target_names** field to make its security names and or authentication domains available to clients. This information may be required by the client to obtain or construct (depending on the mechanism) a suitable initial context token. When the **target_names** field contains more than one name, the names occur in decreasing order of target preference.

[142] Table 16-16 describes the association options that are supported by conforming implementations.

*Table 16-16*EstablishTrustInClient Semantics

	EstablishTrustInClient	**Semantic**
1	Not supported	Target does not support client authentication in service context (at this compound mechanism)
2	Supported	Target supports client authentication in service context. If a CSS does not send an initial context token (in an EstablishContext service context element), then the caller identity is obtained from the transport
3	Required	Target requires client authentication in service context. The CSS may have also authenticated in the transport, but the caller identity is obtained from the service context layer

[143] When a compound mechanism that implements client authentication functionality above the transport also contains a transport mechanism (in the **transport_mech** field), any required association options configured in the transport component shall be interpreted as a prerequisite to satisfying the requirements of the client authentication mechanism.

struct SAS_ContextSec

[144] The SAS_ContextSec structure is used in the **sas_context_mech** field within a CompoundSecMech structure in a TAG_CSI_SEC_MECH_LIST component to describe the security functionality that the target expects to be layered above the transport in service context by means of the **identity_token** and **authorization_token** of the EstablishContext element of the SAS service context protocol. The security functionality represented by this structure is configured as association options in the **target_supports** and **target_requires** fields.

```
typedef const short ServiceConfigurationSyntax;

const ServiceConfigurationSyntax SCS_GeneralNames = 0;
const ServiceConfigurationSyntax SCS_GSSExportedName = 1;

typedef sequence <octet> ServiceSpecificName;

// The name field of the ServiceConfiguration structure identifies a privilege
// authority in the format identified in the syntax field. If the syntax is
// SCS_GeneralNames, the name field contains an ASN.1 (BER) SEQUENCE
// [1..MAX] OF GeneralName, as defined by the type GeneralNames in
// [IETF RFC 2459]. If the syntax is SCS_GSSExportedName, the name field
// contains a GSS exported name encoded according to the rules in
// [IETF RFC 2743] Section 3.2, "Mechanism-Independent Exported Name
// Object Format," p. 84.

struct ServiceConfiguration {
   ServiceConfigurationSyntax syntax;
   ServiceSpecificName name;
};

struct SAS_ContextSec{
   Security::AssociationOptions target_supports;
   Security::AssociationOptions target_requires;
   sequence <ServiceConfiguration> privilege_authorities;
   sequence <Security::OID> supported_naming_mechanisms;
};
```

[145] The **privilege_authorities** field contains a list of the names of 0 or more privilege authorities in decreasing order of target preference. A non-empty list indicates that the target supports authorization tokens provided by a CSS (in EstablishContext messages) and acquired from a named privilege authority. Each ServiceConfiguration element names a privilege authority acceptable to the target. The **syntax** field within the ServiceConfiguration element identifies the format used to represent the authority. Two alternative formats are currently defined: an ASN.1 encoding of the GeneralNames (as defined in [IETF RFC 2459]) which identify a privilege authority, or a GSS exported name (as defined in [IETF RFC 2743] Section 3.2) encoding of the name of a privilege authority.

[146] The **supported_naming_mechanisms** field contains a list of GSS mechanism OIDs. A TSS shall include in this field the complete set of GSS mechanisms for which the target supports (that is, evaluates) identity assertions using an identity token of type ITTPrincipalName. The complete set of identity token types is defined in Section 16.2.5, “Identity Token Format,” on page 16-13.

[147] Table 16-17 describes the combinations of association options that are supported by conforming implementations. Each combination in the table describes the attribute layer functionality of a target that may be defined in a mechanism definition. A target that defines multiple mechanisms may support multiple combinations.

[148] A compound mechanism definition with the DelegationByClient bit set shall include the name of at least one authority in the **privilege_authorities** field.

[149] When a compound mechanism configuration that defines SAS attribute layer functionality also defines client authentication layer or transport layer functionality, any required association options configured in these other layers shall be interpreted as a prerequisite to satisfying the requirements of the functionality defined in the attribute layer

*Table 16-17*Attribute Layer Association Option Combinations

	DelegationByClient	**IdentityAssertion**	**Semantic**
1	Not supported	Not supported	Target does not support identity assertion (that is, identity tokens in the EstablishContext message of the SAS protocol). The caller identity will be obtained from the authentication layer(s).
2	Not supported	Supported	Target evaluates asserted caller identities based on trust rules of the target. In the absence of an asserted identity, the caller identity will be obtained from the authentication layer(s).
3	Supported	Not supported	Target accepts delegation tokens that indicate who has been endorsed to assert an identity. Target does not accept asserted caller identities. The caller identity will be obtained from the authentication layer(s).
4	Supported	Supported	Target accepts delegation tokens that indicate who has been endorsed to assert an identity. Target evaluates asserted caller identities based on trust rules of the target or based on endorsements in a delegation token. In the absence of an asserted identity, the caller identity will be obtained from the authentication layer(s).
5	Required	Not supported	Same as 3, with the addition that target requires a delegation token that endorses the target as proxy for the caller
6	Required	Supported	Same as 4, with the addition that target requires a delegation token that endorses the target as proxy for the caller

TAG_NULL_TAG

[150] This new tagged component is used in the **transport_mech** field of a CompoundSecMech structure to indicate that the compound mechanism does not implement security functionality at the transport layer.

```
// The body of TAG_NULL_TAG component shall be a sequence of octets of
// length 0
const IOP::ComponentId TAG_NULL_TAG = 34;
```

16.5.2 Client-side Mechanism Selection

[151] A client should evaluate the compound security mechanism definitions contained within the CompoundSecMechList in the TAG_CSI_SEC_MECH_LIST component in an IOR to select a mechanism that supports the options required by the client.

[152] The options supported by a compound mechanism are the union (the logical OR) of the options supported by the **transport_mech**, **as_context_mech**, and **sas_context_mech** fields of the CompoundSecMech structure.

[153] The following table defines the semantics defined by the union of association options in compound mechanism definitions. Assoiation options for server to client authentication and message protection add additional semantics that are not represented in the table.

*Table 16-18*Interpretation of Compound Mechanism Association Options

	Semantic	**EstablishTrustInClient**		**IdentityAssertion**	**DelegationByClient**	
		Supported	**Required**	**Supported**	**Supported**	**Required**
1	No client identification				Don’t care[2]	
2	Presumed trust			X		
3	Authentication optional	X			Don’t care	
4	Authentication optional, assertion supported	X		X		
5	Authentication Required	X	X		Don’t care	
6	Authentication Required, assertion supported	X	X	X		
7	Presumed trust including support for provided target restrictions			X	X	
8	Authentication optional, assertion supported including forward trust rules	X		X	X	
9	Authentication required, assertion supported including forward trust rules	X	X	X	X	
10	Presumed Trust including support for provided target restrictions, delegation token required which implies assertion required[1]			X	X	X
11	Authentication optional, assertion supported including forward trust rules, delegation token required which implies either client authentication or assertion required	X		X	X	X
12	Authentication required, delegation token required	X	X		X	X
13	Authentication required, assertion supported including forward trust rules, delegation token required	X	X	X	X	X

1. If a delegation token is required, a non-anonymous client identity shall be established so that it can be endorsed by the delegation token. This same rule applies to row 11, and explains why there is no row that supports client authentication and requires a delegation token.

2. If DelegationByClient is supported, a delegation token may be provided, but it is not required to process the request

16.5.3 Client-Side Requirements and Location Binding

[154] The primary assumption of this interoperability protocol is that transport layer security can ensure that it is not necessary to issue a preliminary request to establish a confidential association with the intended target.

[155] In order to sustain this assumption, trust in target and a confidential transport shall be established prior to issuing any call that may contain arguments (including object keys) or service context elements that the client considers confidential. A CSS acting on behalf of a client may trust a target to locate an object (process a locate request) without having to trust the target with confidential arguments (other than object keys) or service context elements. For example, a CSS may have established a confidential connection to an address it learned from an IOR, and may then determine if the client trusts the target with its request arguments and any associated service context elements. If the client does not trust the target with its request, the CSS may send a locate request.[11] If the locate reply contains a new address, the CSS may establish a new confidential connection, evaluate the level of trust the client has in the new target, and determine whether it can issue the client's request to the target. If in response to the request, the CSS receives a location forward, it will establish another confidential connection with the new address and repeat its trust determination.

[156] Compound security mechanisms appearing in IORs leading to a location daemon should not require clients to authenticate using the username/password mechanism if doing so would cause an overly trusting caller to share its password with an untrusted location daemon.

[157] The way in which a location daemon derives an IOR for a target object is not prescribed by this specification.

Comments on Establishing Trust in Client

[158] A client that does not have the artifacts necessary to provide evidence of its authenticity over at least one of the transports supported by it and its target should search the IOR for a security mechanism definition that does not require client authentication to occur in a transport mechanism.

11. This requires that the CSS be provided with a method to cause the ORB to issue a locate request. There is no standard API to cause an ORB to issue a locate request.

16.6 Conformance Levels

16.6.1 Conformance Level 0

[159] Level 0 defines the base level of secure interoperability that all implementations are required to support. Level 0 requires support for SSL/TLS protected connections. Level 0 implementations are also required to support username/password client authentication and identity assertion by using the service context protocol defined in this specification.

Transport-Layer Requirements

[160] Implementations shall support the Security Attribute Service (SAS) protocol within the service context lists of GIOP request and reply messages exchanged over SSL 3.0 and TLS 1.0 protected connections.

[161] Implementations shall also support the SAS protocol within the service context lists of GIOP request and reply messages over unprotected transports defined within IIOP.[12]

Required Ciphersuites

[162] Conforming implementations are required to support both SSL 3.0 and TLS 1.0 and the mandatory TLS 1.0 ciphersuites identified in [IETF RFC 2246]. Conforming implementations are also required to support the SSL 3.0 ciphersuites corresponding to the mandatory TLS 1.0 ciphersuites.

[163] An additional set of recommended ciphersuites is identified in Section , “Recommended SSL/TLS Ciphersuites,” on page 16-29.

Service Context Protocol Requirements

[164] All implementations shall support the Security Attribute Service (SAS) context element protocol in the manner described in the following sections.

Stateless Mode

[165] All implementations shall support the stateless CSS and stateless TSS modes of operation as defined in Section 16.3.2, “Session Semantics,” on page 16-20, and in the protocol message definitions appearing in Section 16.2.2, “SAS context_data Message Body Types,” on page 16-5.

Client Authentication Tokens and Mechanisms

[166] All implementations shall support the username password (GSSUP) mechanism for client authentication as defined in Section , “Username Password GSS Mechanism (GSSUP),” on page 16-12.

12. SAS protocol elements should only be sent over unprotected transports within trusted environments.

Identity Tokens and Identity Assertion

[167] All implementations shall support the identity assertion functionality defined in Section , “Context Validation,” on page 16-15and the identity token formats and functionality defined in Section 16.2.5, “Identity Token Format,” on page 16-13.

[168] All implementations shall support GSSUP mechanism specific identity tokens of type ITTPrincipalName.

Authorization Tokens (not required)

[169] At this level of conformance, implementations are not required to be capable of including an authorization token in the SAS protocol elements they send or of interpreting such tokens if they are included in received SAS protocol elements.

[170] The format of authorization tokens is defined in Section 16.2.3, “Authorization Token Format,” on page 16-10.

Interoperable Object References (IORs)

[171] The security mechanism configuration of CSIv2 target objects, shall be as defined in Section 16.5.1, “Target Security Configuration,” on page 16-30, with the exception that Level 0 implementations are not required to support the DelegationByClient functionality described in Section , “AssociationOptions Type,” on page 16-31.

16.6.2 Conformance Level 1

[172] Level 1 adds the following additional requirements to those of Level 0.

Authorization Tokens

[173] Level 1 implementations shall support the push model for privilege attributes.

[174] Level 1 requires that a CSS provide clients with an ability to include an authorization token, as defined in Section 16.2.3, “Authorization Token Format,” on page 16-10, in SAS EstablishContext protocol messages.

[175] Level 1 requires that a TSS be capable of evaluating its support for a received authorization token according to the rules defined in Section , “Extensions of the IETF AC Profile for CSIv2,” on page 16-11.

[176] A Level 1 TSS shall recognize the standard attributes and extensions defined in the attribute certificate profile defined in [IETF ID PKIXAC].

[177] Level 1 requires that a target object that supports pushed privilege attributes include in its IORs the names of the privilege authorities trusted by the target object (as defined in Section , “struct SAS_ContextSec,” on page 16-37).

16.6.3 Conformance Level 2

[178] Level 2 adds to Level 1 the following additional requirements.

Authorization-Token-Based Delegation

[179] Level 2 adds to Level 1 a requirement that implementations support the authorization-token-based delegation mechanism implemented by the SAS protocol.

[180] A Level 2 TSS shall be capable of evaluating proxy rules arriving in an authorization token to determine whether an asserting entity has been endorsed (by the authority which vouched for the privilege attributes in the authorization token) to assert the identity to which the privilege attributes pertain. The semantics of the relationship between the identity token and authorization token shall be as defined in Section , "Context Validation," on page 16-15.

[181] A Level 2 TSS shall recognize the "Extensions of the IETF AC Profile for CSIv2"" (that is, the Proxy Info extension) as defined on page 16-11.

[182] Level 2 requires that a target object that accepts identity assertions based on endorsements in authorization tokens represent this support in its IORs as defined in Table 16-17 on page 16-39.

[183] Level 2 requires that a target object that requires an endorsement to act as proxy for its callers represent this requirement in its IORs as defined in Table 16-17 on page 16-39.

16.6.4 Stateful Conformance

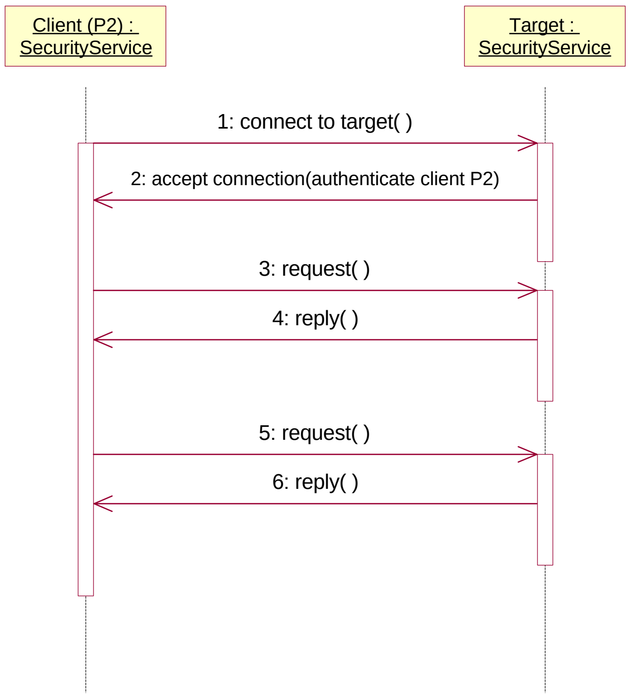
[184] Implementations are differentiated not only by the conformance levels described in the preceding sections but also by whether or not they support stateful security contexts.

[185] For an implementation to claim stateful conformance, it shall implement the stateless and stateful functionality as defined in Section 16.3.2, "Session Semantics," on page 16-20 and in Section 16.2.2, "SAS context_data Message Body Types," on page 16-5.

16.7 Sample Message Flows and Scenarios

[186] This appendix contains sequence diagrams and sample IORs for a set of scenarios selected to illustrate the interoperability protocols defined in this submission. The sample IORs are expressed in pseudocode.

16.7.1 Confidentiality, Trust in Server, and Trust in Client Established in the Connection



1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue. Client (P2) authenticates to TSS in the handshake.
3. Send request (with no security service context element).
4. Receive reply (with no security service context element).
5. Same as 3.
6. Same as 4.

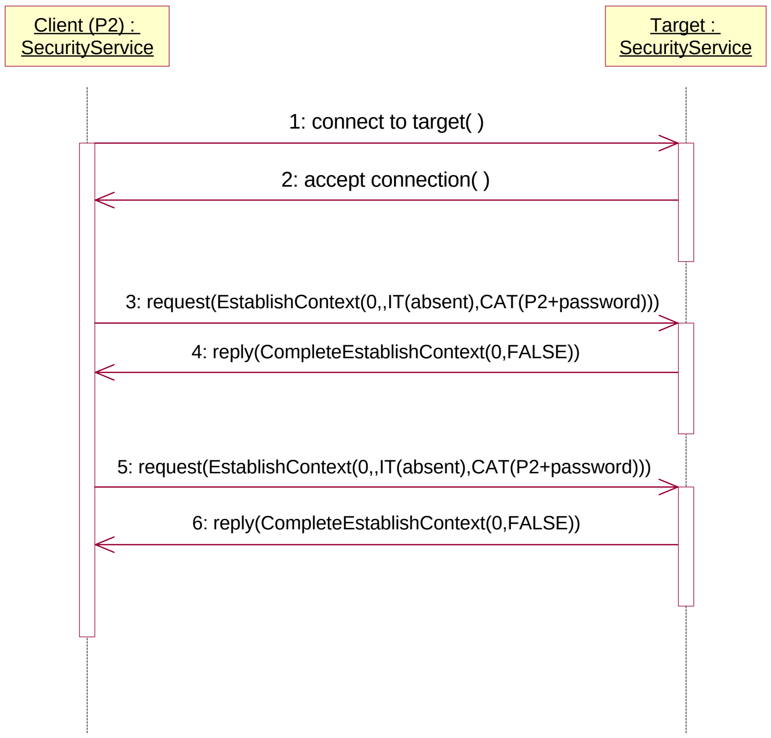
Sample IOR Configuration

[187] The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList {
   stateful = FALSE;
   sequence <CompoundSecMech>  {
      CompoundSecMec {
         target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
         transport_mech = TAG_SSL_SEC_TRANS {
            target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                           EstablishTrustInTarget};
            target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
            port = x;
         };
         as_context_mech = {
            target_supports = {};
            ...
         };
         sas_context_mech = {
            target_supports = {};
            ...
         };
      };
   };
};
```

[188] Note that based on the ciphersuites listed in Section , “Required Ciphersuites,” on page 16-42 and the rules for target_supports and target_requires appearing in the tables in Section , “TAG_SSL_SEC_TRANS,” on page 16-33, all target IORs should include {Integrity, Confidentiality, EstablishTrustInTarget} in target_supports and at least {Integrity, Confidentiality} in target_requires. This statement applies to all the sample IORs corresponding to all the scenarios described in this appendix.

16.7.2 *Confidentiality and Trust in Server Established in the Connection - Stateless Trust in Client Established in Service Context*


Client (P2) : SecurityService
Target : SecurityService
1: connect to target( )
2: accept connection( )
3: request(EstablishContext(0,,IT(absent),CAT(P2+password)))
4: reply(CompleteEstablishContext(0,FALSE))
5: request(EstablishContext(0,,IT(absent),CAT(P2+password)))
6: reply(CompleteEstablishContext(0,FALSE))

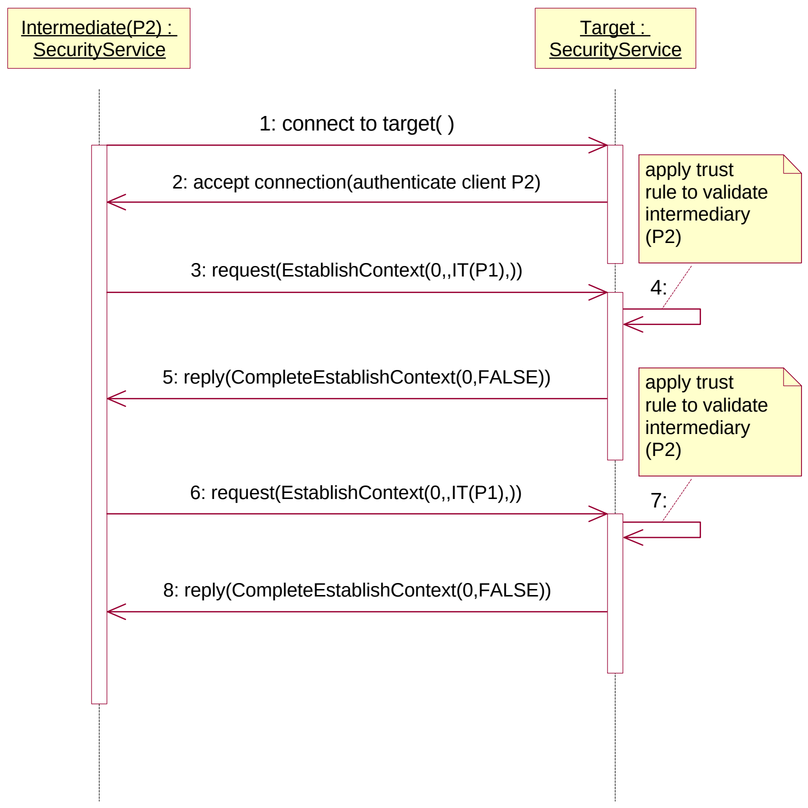
1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue.
3. Send request (with stateless security service context element containing a client_authentication_token).
4. Receive reply with CompleteEstablishContext service context element indicating context (and request) was accepted.
5. Same as 3.
6. Same as 4.

Sample IOR Configuration

[189] The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList {
  stateful = FALSE;
  sequence <CompoundSecMech> {
    CompoundSecMec {
      target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
      transport_mech = TAG_SSL_SEC_TRANS {
        target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                    EstablishTrustInTarget};
        target_requires = {Integrity, Confidentiality};
        port = x;
      };
      as_context_mech = {
        target_supports = {EstablishTrustInClient};
        target_requires = {EstablishTrustInClient};
        client_authentication_mech = GSSUP_OID;
        target_name = (scope = realm);
        ...
      };
      sas_context_mech = {
        target_supports = {};
        ...
      };
    };
  };
};
```

16.7.3 Confidentiality, Trust in Server, and Trust in Client Established in the Connection - Stateless Trust Association Established in Service Context


Intermediate(P2) : SecurityService
Target : SecurityService
1: connect to target( )
2: accept connection(authenticate client P2)
apply trust rule to validate intermediary (P2)
3: request(EstablishContext(0,,IT(P1),))
4:
5: reply(CompleteEstablishContext(0,FALSE))
apply trust rule to validate intermediary (P2)
6: request(EstablishContext(0,,IT(P1),))
7:
8: reply(CompleteEstablishContext(0,FALSE))

1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue. Client (P2) authenticates to TSS in the handshake.
3. Send request (with stateless security service context element containing spoken for identity (P1) in identity_token).
4. TSS validates that target trusts P2 to speak for P1.
5. Receive reply with CompleteEstablishContext service context element indicating context (and request) was accepted.
6. Same as 3.
7. Same as 4.
8. Same as 5.

Sample IOR Configuration

[190] The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList {
  stateful = FALSE;
  sequence <CompoundSecMech> {
    CompoundSecMec {
      target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
      transport_mech = TAG_SSL_SEC_TRANS {
        target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                    EstablishTrustInTarget};
        target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
        port = x;
      };
      as_context_mech = {
        target_supports = {};
        ...
      };
      sas_context_mech = {
        target_supports = {IdentityAssertion};
        ...
      };
    };
  };
};
```

Validating the Trusted Server

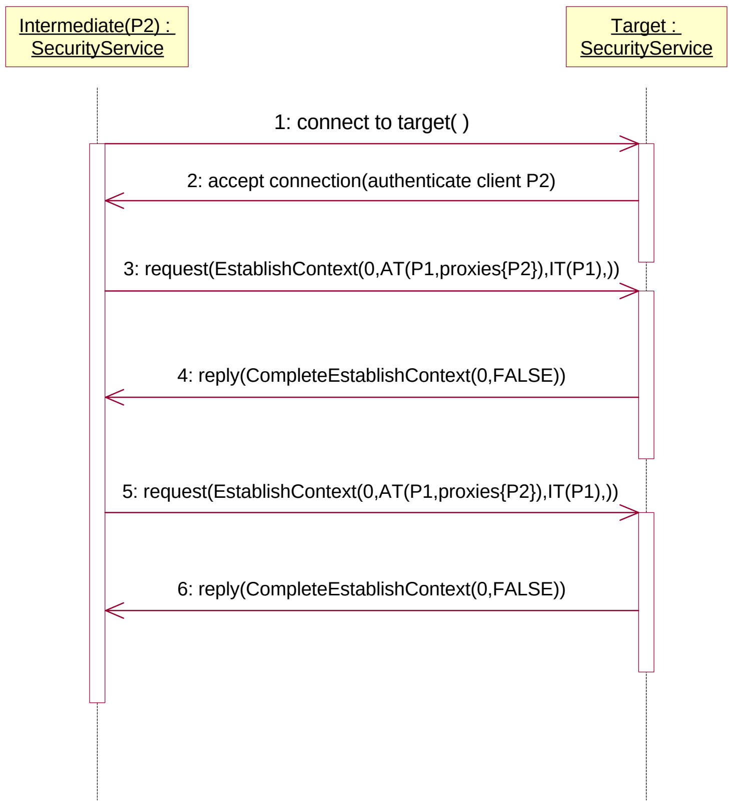
[191] If trust is not presumed, then the TSS shall evaluate the trustworthiness of the speaking for identity (i.e. the client identity established in the authentication layer(s) - P2 in the preceding example) in order to determine if it is authorized to speak for the spoken for identity (i.e. the non-anonymous identity represented as P1 in the identity token in the preceding example).

Presuming the Security of the Connection

[192] There are variants of this scenario where either no security is established in the connection, or the connection is used to establish confidentiality only, and/or trust in the target only. These cases all fall under what is referred to as a presumed trust association. Where the security of the connection and the party using it is presumed, the TSS will not validate the trustworthiness of the speaking-for identity.

```
CompoundSecMechList {
    stateful = FALSE;
    sequence <CompoundSecMech> {
        CompoundSecMec {
            target_requires = {Integrity, Confidentiality};
            transport_mech = TAG_SSL_SEC_TRANS {
                target_supports = {Integrity, Confidentiality,
                          EstablishTrustInTarget};
                target_requires = {Integrity, Confidentiality};
                port = x;
            };
            as_context_mech = {
                target_supports = {};
                ...
            };
            sas_context_mech = {
                target_supports = {IdentityAssertion};
                ...
            };
        };
    };
};
```

16.7.4 Confidentiality, Trust in Server, and Trust in Client Established in the Connection - Stateless Forward Trust Association Established in Service Context


Intermediate(P2) : SecurityService
Target : SecurityService
1: connect to target( )
2: accept connection(authenticate client P2)
3: request(EstablishContext(0,AT(P1,proxies{P2}),IT(P1),))
4: reply(CompleteEstablishContext(0,FALSE))
5: request(EstablishContext(0,AT(P1,proxies{P2}),IT(P1),))
6: reply(CompleteEstablishContext(0,FALSE))

1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue. Intermediate (P2) authenticates to TSS in the handshake.
3. Send request with stateless security service context element containing spoken for identity (P1) in identity_token, and trust rule from P1 in authorization_token delegating proxy to P2.
4. Receive reply with CompleteEstablishContext service context element indicating context (and request) was accepted.
5. Same as 3.
6. Same as 4.

Sample IOR Configuration

[193] The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList {
   stateful = FALSE;
   sequence <CompoundSecMech> {
      CompoundSecMec {
         target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
         transport_mech = TAG_SSL_SEC_TRANS {
            target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                           EstablishTrustInTarget};
            target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
            port = x;
         };
         as_context_mech = {
            target_supports = {};
            ...
         };
         sas_context_mech = {
            target_supports = {IdentityAssertion, DelegationByClient};
            ...
         };
      };
   };
};
```

16.8 *References for this Chapter*

CORBASEC

CORBA Security Service, Revision 1.2, http://www.omg.org/docs/ptc/98-01-02

CORBA Security Service, Revision 1.5, http://www.omg.org/docs/ptc/98-12-03

CORBA Security Service, Revision 1.7, http://www.omg.org/docs/ptc/99-12-03

IETF ID PKIXAC	An Internet Attribute Certificate Profile for Authorization, <draft-ietf-pkix-ac509prof-05.txt>, S. Farrell, Baltimore Technologies, R. Housley, SPYRUS, August 2000.
IETF RFC 2246	The TLS Protocol Version 1.0, T. Dierks, C. Allen, January 1999.
IETF RFC 2459	Internet X.509 Public Key Infrastructure Certificate and CRL Profile, R Housley, W. Ford, W. Polk, and D. Solo, January 1999.
IETF RFC 2743	Generic Security Service Application Program Interface Version 2, Update 1, J. Linn, January 2000.
X.501-93	ITU-T Recommendation X.501: Information Technology - Open Systems Interconnection - The Directory: Models, 1993.

16.9 IDL

16.9.1 Module IOP

New Types Defined for CSIv2

const ServiceId SecurityAttributeService = 15;

16.9.2 Module Security

[194] This subsection defines the OMG IDL for security data types common to the other security modules, which is the module **Security**. The **Security** module depends on the **TimeBase** module and the **CORBA** module.

```
#if !defined(_SECURITY_IDL_)
#define _SECURITY_IDL_
#include <orb.idl>
#include <TimeBase.idl>
#pragma prefix "omg.org"

module Security {

#     pragma version Security 1.5

      typedef string                        SecurityName;
      typedef sequence <octet>              Opaque;

      // Constant declarations for Security Service Options

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

      // Service options for Common Secure Interoperability

      const CORBA::ServiceOption CommonInteroperabilityLevel0 = 10;
      const CORBA::ServiceOption CommonInteroperabilityLevel1 = 11;
      const CORBA::ServiceOption CommonInteroperabilityLevel2 = 12;

      // Security mech types supported for secure association
      const CORBA::ServiceDetailType SecurityMechanismType = 1;

      // privilege types supported in standard access policy
      const CORBA::ServiceDetailType SecurityAttribute = 2;

      // extensible families for standard data types

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

      // security attributes

      typedef unsigned long           SecurityAttributeType;

      // other attributes; family = 0

      const SecurityAttributeType             AuditId = 1;
      const SecurityAttributeType             AccountingId = 2;
      const SecurityAttributeType             NonRepudiationId = 3;
```

```
// privilege attributes; family = 1

const SecurityAttributeType              _Public = 1;
const SecurityAttributeType              AccessId = 2;
const SecurityAttributeType              PrimaryGroupId = 3;
const SecurityAttributeType              GroupId = 4;
const SecurityAttributeType              Role = 5;
const SecurityAttributeType              AttributeSet = 6;
const SecurityAttributeType              Clearance = 7;
const SecurityAttributeType              Capability = 8;

struct AttributeType {
    ExtensibleFamily               attribute_family;
    SecurityAttributeType          attribute_type;
};

typedef sequence<AttributeType>           AttributeTypeList;

struct SecAttribute {
    AttributeType              attribute_type;
    Opaque                     defining_authority;
    Opaque                     value;
    // the value of this attribute can be
    // interpreted only with knowledge of type
};

typedef sequence <SecAttribute> AttributeList;

// Authentication return status

enum AuthenticationStatus {
    SecAuthSuccess,
    SecAuthFailure,
    SecAuthContinue,
    SecAuthExpired
};

// Association return status

enum AssociationStatus {
    SecAssocSuccess,
    SecAssocFailure,
    SecAssocContinue
};

// Authentication method
typedef unsigned long AuthenticationMethod;
typedef sequence<AuthenticationMethod> AuthenticationMethodList;

// Credential types which can be set as Current default

enum CredentialType {
    SecInvocationCredentials,
    SecNRCredentials
};

enum InvocationCredentialsType {
    SecOwnCredentials,
    SecReceivedCredentials
};

// Declarations related to Rights

struct Right {
    ExtensibleFamily        rights_family;
    string                  right;
};

typedef sequence <Right> RightsList;

enum RightsCombinator {
    SecAllRights,
    SecAnyRight
};

// Delegation related

enum DelegationState {
    SecInitiator,
    SecDelegate
};

enum DelegationDirective {
    Delegate,
    NoDelegate
};

// pick up from TimeBase

typedef TimeBase::UtcT          UtcT;
typedef TimeBase::IntervalT     IntervalT;
typedef TimeBase::TimeT         TimeT;
```

```
// Security features available on credentials.

enum SecurityFeature {
      SecNoDelegation,
      SecSimpleDelegation,
      SecCompositeDelegation,
      SecNoProtection,
      SecIntegrity,
      SecConfidentiality,
      SecIntegrityAndConfidentiality,
      SecDetectReplay,
      SecDetectMisordering,
      SecEstablishTrustInTarget,
      SecEstablishTrustInClient
};

// Quality of protection which can be specified
// for an object reference and used to protect messages

enum QOP {
      SecQOPNoProtection,
      SecQOPIntegrity,
      SecQOPConfidentiality,
      SecQOPIntegrityAndConfidentiality
};

// Type of SecurityContext

enum SecurityContextType {
      SecClientSecurityContext,
      SecServerSecurityContext
};

// Operational State of a Security Context

enum SecurityContextState {
      SecContextInitialized,
      SecContextContinued,
      SecContextClientEstablished,
      SecContextEstablished,
      SecContextEstablishExpired,
      SecContextExpired,
      SecContextInvalid
};

// For use with SecurityReplaceable

struct OpaqueBuffer {
      Opaque                    buffer;
      unsigned long             startpos;
      unsigned long             endpos;
      // startpos <= endpos
      // OpaqueBuffer is said to be empty if startpos == endpos
};
```

```
// Association options which can be administered
// on secure invocation policy and used to
// initialize security context

typedef unsigned short              AssociationOptions;

const AssociationOptions NoProtection = 1;
const AssociationOptions Integrity = 2;
const AssociationOptions Confidentiality = 4;
const AssociationOptions DetectReplay = 8;
const AssociationOptions DetectMisordering = 16;
const AssociationOptions EstablishTrustInTarget = 32;
const AssociationOptions EstablishTrustInClient = 64;
const AssociationOptions NoDelegation = 128;
const AssociationOptions SimpleDelegation = 256;
const AssociationOptions CompositeDelegation = 512;
const AssociationOptions IdentityAssertion = 1024;
const AssociationOptions DelegationByClient = 2048;
```

```
//Types Defined for CSIv2

typedef sequence <octet> OID;

// An X509CertificateChain contains an ASN.1 BER encoded SEQUENCE
// [1..MAX] OF  X.509 certificates encapsulated in a sequence of octets. The
// subject’s certificate shall come first in the list. Each following certificate shall
// directly certify the one preceding it. The ASN.1 representation of Certificate is
// as defined in [IETF RFC 2459].

typedef sequence <octet> X509CertificateChain;

// an X.501 type name or Distinguished Name encapsulated in a sequence of
// octets containing the ASN.1 encoding.

typedef sequence <octet> X501DistinguishedName;

typedef sequence <octet> UTF8String;

typedef UTF8String NameValue;

struct ScopedName {
      Security::NameValue name_scope;
      Security::NameValue name_value;
};

// A sequence of octets containing a GSStoken. Initial context tokens are ASN.1
// encoded as defined in [IETF RFC 2743] Section 3.1, "Mechanism-Independent
// token Format", pp. 81-82. Initial context tokens contain an ASN.1 tag followed
// by a token length, a mechanism identifier, and a mechanism-specific token
// (i.e. a GSSUP::InitialContextToken). The encoding of all other GSS tokens (e.g.
// error tokens and final context tokens) is mechanism dependent.

typedef sequence <octet> GSSToken;

// An encoding of a GSS Mechanism-Independent Exported Name Object as
// defined in [IETF RFC 2743] Section 3.2, "GSS Mechanism-Independent
// Exported Name Object Format," p. 84.

typedef sequence <octet> GSS_NT_ExportedName;

// End types defined for CSIv2

// Flag to indicate whether association options being
// administered are the “required” or “supported” set

enum RequiresSupports {
      SecRequires,
      SecSupports
};
```

```
// Direction of communication for which
// secure invocation policy applies

enum CommunicationDirection {
      SecDirectionBoth,
      SecDirectionRequest,
      SecDirectionReply
};

// security association mechanism type

typedef string                  MechanismType;

typedef sequence<MechanismType> MechanismTypeList;

struct SecurityMechanismData {
      MechanismType             mechanism;
      Opaque                    security_name;
      AssociationOptions        options_supported;
      AssociationOptions        options_required;
};

typedef sequence<SecurityMechanismData>SecurityMechanismDataList;

// AssociationOptions-Direction pair

struct OptionsDirectionPair {
      AssociationOptions        options;
      CommunicationDirection    direction;
};

typedef sequence <OptionsDirectionPair> OptionsDirectionPairList;

// Delegation mode which can be administered

enum DelegationMode {
      SecDelModeNoDelegation,              // i.e. use own credentials
      SecDelModeSimpleDelegation,          // delegate received credentials
      SecDelModeCompositeDelegation        // delegate both;
};

// Association options supported by a given mech type

struct MechandOptions {
      MechanismType             mechanism_type;
      AssociationOptions        options_supported;
};
```

```
typedef sequence <MechandOptions>           MechandOptionsList;

// Attribute of the SecurityLevel2::EstablishTrustPolicy

struct EstablishTrust {
    boolean trust_in_client;
    boolean trust_in_target;
};

// Audit

typedef unsigned long                       AuditChannelId;

typedef unsigned short                      EventType;

const EventType           AuditAll = 0;
const EventType           AuditPrincipalAuth = 1;
const EventType           AuditSessionAuth = 2;
const EventType           AuditAuthorization = 3;
const EventType           AuditInvocation = 4;
const EventType           AuditSecEnvChange = 5;
const EventType           AuditPolicyChange = 6;
const EventType           AuditObjectCreation = 7;
const EventType           AuditObjectDestruction = 8;
const EventType           AuditNonRepudiation = 9;

enum DayOfTheWeek {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

enum AuditCombinator {
    SecAllSelectors,
    SecAnySelector
};

struct AuditEventType {
    ExtensibleFamily          event_family;
    EventType                 event_type;
};
typedef sequence <AuditEventType>           AuditEventTypeList;

typedef unsigned long                       SelectorType;

const SelectorType        InterfaceName = 1;
const SelectorType        ObjectRef = 2;
const SelectorType        Operation = 3;
const SelectorType        Initiator = 4;
const SelectorType        SuccessFailure = 5;
const SelectorType        Time = 6;
const SelectorType        DayOfWeek = 7;
```

```
    // values defined for audit_needed and audit_write are:
    // InterfaceName: CORBA::RepositoryId
    // ObjectRef: object reference
    // Operation: op_name
    // Initiator: Credentials
    // SuccessFailure: boolean
    // Time: utc time on audit_write; time picked up from
    //                  environment in audit_needed if required
    // DayOfWeek: DayOfTheWeek

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

    // Constant declaration for valid Security Policy Types

    // General administrative policies
    const CORBA::PolicyType SecClientInvocationAccess = 1;
    const CORBA::PolicyType SecTargetInvocationAccess = 2;
    const CORBA::PolicyType SecApplicationAccess = 3;
    const CORBA::PolicyType SecClientInvocationAudit = 4;
    const CORBA::PolicyType SecTargetInvocationAudit = 5;
    const CORBA::PolicyType SecApplicationAudit = 6;
    const CORBA::PolicyType SecDelegation = 7;
    const CORBA::PolicyType SecClientSecureInvocation = 8;
    const CORBA::PolicyType SecTargetSecureInvocation = 9;
    const CORBA::PolicyType SecNonRepudiation = 10;

    // Policies used to control attributes of a binding to a target
    const CORBA::PolicyType SecMechanismsPolicy = 12;
    const CORBA::PolicyType SecInvocationCredentialsPolicy = 13;
    const CORBA::PolicyType SecFeaturePolicy = 14; // obsolete
    const CORBA::PolicyType SecQOPPolicy = 15;

    const CORBA::PolicyType SecDelegationDirectivePolicy = 38;
    const CORBA::PolicyType SecEstablishTrustPolicy = 39;
};
#endif /* _SECURITY_IDL_ */
```

16.9.3 Module SSLIOP

[195] The **SSLIOP** module holds the structure and TAG definitions needed for using SSL as the secure transport under CORBA Security. This module depends on the **Security** and the **IOP** modules.

```
#if !defined(_SSLIOP_IDL)
#define _SSLIOP_IDL
#pragma prefix "omg.org"
#include <IOP.idl>
#include<Security.idl>

module SSLIOP {
    // Security mechanism SSL

    const IOP::ComponentId TAG_SSL_SEC_TRANS = 20;

    struct SSL {
        Security::AssociationOptions        target_supports;
        Security::AssociationOptions        target_requires;
        unsigned short                      port;
    };
};
#endif /* _SSLIOP_IDL */
```

16.9.4 Secure Inter-ORB Protocol (SECIOP)

[196] The **SECIOP** module holds structure declarations related to the layout of message fields in the Secure Inter-ORB protocol. This module depends on the **IOP** and **Security** modules.

```
#if !defined(_SECIOP_IDL_)
#define _SECIOP_IDL
#include <IOP.idl>
#include <Security.idl>
#pragma prefix "omg.org"


module SECIOP {

    const IOP::ComponentId TAG_GENERIC_SEC_MECH = 22;

    const IOP::ComponentId TAG_ASSOCIATION_OPTIONS = 13;

    const IOP::ComponentId TAG_SEC_NAME = 14;

    struct TargetAssociationOptions{
            Security::AssociationOptions        target_supports;
            Security::AssociationOptions        target_requires;
    };

    struct GenericMechanismInfo {
            sequence <octet>                    security_mechanism_type;
            sequence <octet>                    mech_specific_data;
            sequence <IOP::TaggedComponent>  components;
    };
```

```
enum MsgType {
        MTEstablishContext,
        MTCompleteEstablishContext,
        MTContinueEstablishContext,
        MTDiscardContext,
        MTMessageError,
        MTMessageInContext
};

typedef unsigned long long ContextId;

enum ContextIdDefn {
        CIDClient,
        CIDPeer,
        CIDSender
};

struct EstablishContext {
        ContextId               client_context_id;
        sequence <octet>        initial_context_token;
};

struct CompleteEstablishContext {
        ContextId               client_context_id;
        boolean                 target_context_id_valid;
        ContextId               target_context_id;
        sequence <octet>        final_context_token;
};

struct ContinueEstablishContext {
        ContextId               client_context_id;
        sequence <octet>        continuation_context_token;
};

struct DiscardContext {
        ContextIdDefn           message_context_id_defn;
        ContextId               message_context_id;
        sequence <octet>        discard_context_token;
};

struct MessageError {
        ContextIdDefn           message_context_id_defn;
        ContextId               message_context_id;
        long                    major_status;
        long                    minor_status;
};

enum ContextTokenType {
        SecTokenTypeWrap,
        SecTokenTypeMIC
};
```

```
struct MessageInContext {
            ContextIdDefn           message_context_id_defn;
            ContextId               message_context_id;
            ContextTokenType        message_context_type;
            sequence <octet>        message_protection_token;
};

// message_protection_token is obtained by CDR encoding
// the following SequencingHeader followed by the octets of the
// frame data. SequencingHeader + Frame Data is called a
// SequencedDataFrame

struct SequencingHeader {
      octet                   control_state;
      unsigned long           direct_sequence_number;
      unsigned long           reverse_sequence_number;
      unsigned long           reverse_window;
};

typedef sequence <octet> SecurityName;
typedef unsigned short CryptographicProfile;
typedef sequence <CryptographicProfile> CryptographicProfileList;

// Cryptographic profiles for SPKM

const CryptographicProfile        MD5_RSA = 20;
const CryptographicProfile        MD5_DES_CBC = 21;
const CryptographicProfile        DES_CBC = 22;
const CryptographicProfile        MD5_DES_CBC_SOURCE  = 23;
const CryptographicProfile        DES_CBC_SOURCE  = 24;

// Security Mechanism SPKM_1

const IOP::ComponentId            TAG_SPKM_1_SEC_MECH = 15;

struct SPKM_1 {
      Security::AssociationOptions           target_supports;
      Security::AssociationOptions           target_requires;
      CryptographicProfileList               crypto_profile;
      SecurityName                           security_name;
};

// Security Mechanism SPKM_1

const IOP::ComponentId TAG_SPKM_2_SEC_MECH = 16;

struct SPKM_2 {
      Security::AssociationOptions           target_supports;
      Security::AssociationOptions           target_requires;
      CryptographicProfileList               crypto_profile;
      SecurityName                           security_name;
};
```

```
// Cryptographic profiles for GSS Kerberos Protocol

const CryptographicProfile              DES_CBC_DES_MAC = 10;
const CryptographicProfile              DES_CBC_MD5 = 11;
const CryptographicProfile              DES_MAC = 12;
const CryptographicProfile              MD5 = 13;

// Security Mechanism KerberosV5

const IOP::ComponentId TAG_KerberosV5_SEC_MECH = 17;

struct KerberosV5 {
    Security::AssociationOptions            target_supports;
    Security::AssociationOptions            target_requires;
    CryptographicProfileList                crypto_profile;
    SecurityName                            security_name;
};

// Cryptographic profiles for CSI-ECMA Protocol

const CryptographicProfile              FullSecurity = 1;
const CryptographicProfile              NoDataConfidentiality = 2;
const CryptographicProfile              LowGradeConfidentiality = 3;
const CryptographicProfile              AgreedDefault = 5;

// Security Mechanism CSI_ECMA_Secret

const IOP::ComponentId TAG_CSI_ECMA_Secret_SEC_MECH = 18;

struct CSI_ECMA_Secret {
    Security::AssociationOptions            target_supports;
    Security::AssociationOptions            target_requires;
    CryptographicProfileList                crypto_profile;
    SecurityName                            security_name;
};

// Security Mechanism CSI_ECMA_Hybrid

const IOP::ComponentId TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;

struct CSI_ECMA_Hybrid {
    Security::AssociationOptions            target_supports;
    Security::AssociationOptions            target_requires;
    CryptographicProfileList                crypto_profile;
    SecurityName                            security_name;
};
```

```
    // Security Mechanism CSI_ECMA_Public

    const IOP::ComponentId TAG_CSI_ECMA_Public_SEC_MECH = 21;

    struct CSI_ECMA_Public {
        Security::AssociationOptions        target_supports;
        Security::AssociationOptions        target_requires;
        CryptographicProfileList            crypto_profile;
        SecurityName                        security_name;
    };

    // Tagged component for configuring SECIOP as a CSIv2 mechanism transport

    const IOP::ComponentId TAG_SECIOP_SEC_TRANS = 35;

    struct SECIOP_SEC_TRANS {
            Security::AssociationOptions target_supports;
            Security::AssociationOptions target_requires;
            Security::OID mech_oid;
            Security::GSS_NT_ExportedName target_name;
            unsigned short port;
    };
};
#endif /* _SECIOP_IDL */
```

16.9.5 Module GSSUP - Username/Password GSSAPI Token Formats

```
module GSSUP {

    // The GSS Object Identifier allocated for the username/password
    // mechanism is defined below.
    //
    //   { iso-itu-t (2) international-organization (23) omg (130) security (1)
    //     authentication (1) gssup-mechanism (1) }

    // The following structure defines the inner contents of the username
    // password initial context token. This structure is CDR encoded in a
    // sequence of octets and appended at the end of the username/password
    // GSS (initial context) Token (see above).

    struct InitialContextToken {
            Security::ScopedName username;
            Security::UTF8String password;
    };

}; // GSSUP
```

16.9.6 Module CSI - Common Secure Interoperability

```
module CSI {

    // The MsgType enumeration defines the complete set of service context
    // message types used by the CSI context management protocols, including
    // those message types pertaining only to the stateful application of the
    // protocols (to insure proper alignment of the identifiers between stateless
    // and stateful implementations). Specifically, the MTMessageInContext is
    // not sent by stateless clients (although it may be received by stateless
    // targets).

    typedef short MsgType;

    const MsgType MTEstablishContext = 0;
    const MsgType MTCompleteEstablishContext = 1;
    const MsgType MTContextError = 4;
    const MsgType MTMessageInContext = 5;

    // The ContextId type is used carry session identifiers. A stateless
    // application of the service context protocol is indicated by a session
    // identifier value of 0.

    typedef unsigned long long ContextId;

    // The AuthorizationElementType defines the contents and encoding of
    // the_element field of the AuthorizationElement.

    typedef unsigned long AuthorizationElementType;

    // An AuthorizationElementType of X509AttributeCertChain indicates that
    // the_element field of the AuthorizationElement contains an ASN.1 BER
    // SEQUENCE composed of an (X.509) AttributeCertificate followed by a
    // SEQUENCE OF (X.509) Certificate. The two-part SEQUENCE is encapsu-
    // lated in an octet stream. The chain of identity certificates is provided to
    // certify the attribute certificate. Each certificate in the chain shall directly
    // certify the one preceding it. The first certificate in the chain shall certify
    // the attribute certificate. The ASN.1 representation of (X.509) Certificate is
    // as defined in [IETF RFC 2459]. The ASN.1 representation of (X.509)
    // AtributeCertificate is as defined in [IETF ID PKIXAC].

    const AuthorizationElementType X509AttributeCertChain = 1;

    // The AuthorizationElement contains one element of an authorization token.
    // Each element of an authorization token is logically a PAC.

    struct AuthorizationElement {
            AuthorizationElementType   the_type;
            sequence <octet>           the_element;
    };

    // The AuthorizationToken is made up of a sequence of
    // AuthorizationElements

    typedef sequence <AuthorizationElement> AuthorizationToken;
typedef short IdentityTokenType;

const IdentityTokenType ITTAbsent = 0;
const IdentityTokenType ITTAnonymous = 1;
const IdentityTokenType ITTPrincipalName = 2;
const IdentityTokenType ITTX509CertChain = 3;
const IdentityTokenType ITTDistinguishedName = 4;

union IdentityToken switch ( IdentityTokenType ) {
        case ITTAbsent: boolean absent;
        case ITTAnonymous: boolean anonymous;
        case ITTPrincipalName: Security::GSS_NT_ExportedName
            principal_name;
        case ITTX509CertChain: Security::X509CertificateChain
            certificate_chain;
        case ITTDistinguishedName: Security::X501DistinguishedName dn;
};

struct EstablishContext {
        CSI::ContextId client_context_id;
        AuthorizationToken authorization_token;
        IdentityToken identity_token;
        Security::GSSToken client_authentication_token;
};

struct CompleteEstablishContext {
        CSI::ContextId client_context_id;
        boolean context_stateful;
        Security::GSSToken final_context_token;
};

struct ContextError {
        CSI::ContextId client_context_id;
        long major_status;
        long minor_status;
        Security::GSSToken error_token;
};

// Not sent by stateless clients. If received by a stateless server, a
// ContextError message should be returned, indicating the session does
// not exist.

struct MessageInContext {
        CSI::ContextId client_context_id;
        boolean discard_context;
};

union SASContextBody switch ( CSI::MsgType ) {
        case CSI::MTEstablishContext: EstablishContext establish_msg;
        case CSI::MTCompleteEstablishContext: CompleteEstablishContext
            complete_msg;
        case CSI::MTContextError: ContextError error_msg;
        case CSI::MTMessageInContext: MessageInContext in_context_msg;
};
```

```
}; // CSI
```

16.9.7 CSIIOP - CSIv2 IOR Component Tag Definitions

```
module CSIIOP {

    const IOP::ComponentId TAG_NULL_TAG = 34;
    const IOP::ComponentId TAG_CSI_SEC_MECH_LIST = 33;

    typedef short ServiceConfigurationSyntax;

    const ServiceConfigurationSyntax SCS_GeneralNames = 0;
    const ServiceConfigurationSyntax SCS_GSSExportedName = 1;

    typedef sequence <octet> ServiceSpecificName;

    // The name field of the ServiceConfiguration structure identifies a privilege
    // authority in the format identified in the syntax field. If the syntax is
    // SCS_GeneralNames, the name field contains an ASN.1 (BER) SEQUENCE
    // [1..MAX] OF GeneralName, as defined by the type GeneralNames in
    // [IETF RFC 2459]. If the syntax is SCS_GSSExportedName, the name field
    // contains a GSS exported name encoded according to the rules in
    // [IETF RFC 2743] Section 3.2, "Mechanism-Independent Exported Name
    // Object Format," p. 84.

    struct ServiceConfiguration {
            ServiceConfigurationSyntax syntax;
            ServiceSpecificName name;
    };

    // The body of the TAG_NULL_TAG component is a sequence of octets of
    // length 0.

    // type used to define AS layer functionality within a compound mechanism
    // definition

    struct AS_ContextSec{
            Security::AssociationOptions target_supports;
            Security::AssociationOptions target_requires;
            Security::OID client_authentication_mech;
            sequence <Security::GSS_NT_ExportedName> target_name;
    };

    // type used to define SAS layer functionality within a compound mechanism
    // definition

    struct SAS_ContextSec{
            Security::AssociationOptions target_supports;
            Security::AssociationOptions target_requires;
            sequence <ServiceConfiguration> privilege_authorities;
            sequence <Security::OID> supported_naming_mechanisms;
    };

    // type used in the body of a TAG_CSI_SEC_MECH_LIST component to
    // describe a compound mechanism

    struct CompoundSecMech {
```

```
                Security::AssociationOptions target_requires;
                IOP::TaggedComponent transport_mech;
                AS_ContextSec as_context_mech;
                SAS_ContextSec sas_context_mech;
        };

        // type corresponding to the body of a TAG_CSI_SEC_MECH_LIST
        // component

        struct CompoundSecMechList {
                boolean stateful;
                sequence <CompoundSecMech> mechanism_list;
        };

    }; //CSIIOP
```