ORBASEC SL2

User Guide

Adiron, LLC
2-212 CST
CASE Center
Syracuse University
Syracuse, NY 13244-4100

Version 2.0
April 1999

CHAPTER 1

ORBASEC SL2 Introduction

What is ORBASEC SL2?

THIS IS THE RELEASE 2.0 VERSION OF ORBASEC SL2

ORBASEC SL2 is a secure Object Request Broker (ORB) that is compliant with the Common Object Request Broker Architecture (CORBA) security service specification as defined by the Object Management Group (OMG). ORBASEC SL2 is compliant with the Security Level 2 specification of the proposed 1.6 Revision of the Security Service Specification [4].

The features ORBASEC SL2 supports:

- Full functionality of the ORB, ORBACUS 3.1.2 for Java.[6]
- CORBA Security Level 2 Functionality.
- SECIOP - SECure Inter-Operability Protocol compliant.
- Security Replaceability
- Kerberos Version 5 (GSS-API)
- Secure Sockets Layer Version 3 (SSL)
- Unprotected Communication (IIOP)

ORBASEC SL2 gives the application developer the means necessary to provide security in the form of authentication and strongly encrypted messsaging to write develop and deploy secure distributed applications.

Developments Since ORBASEC SL2 Beta 3

During the ORBASEC SL2 Beta 1 through Beta 3 phases, work was going on at the OMG to update the security specification. Some of the interfaces that were presented in ORBASEC SL2 in those phases were listed as extensions to the Security Level 2 specification. We are now happy to report that most of those interfaces are no longer extensions to the Security Level 2 specification, but are now part of it. This document can be compared with the Beta 2 document for the differences.

In Beta 3, we have updated the GSS-API portion of the Kerberos implementation from MIT. These modifications fixed some problems with the library, concerning bad handling of delegated credentials. Also, ORBAsec SL2 now has the ability to use memory based credentials caches, so that a client's credentials do not have to reside on a file on the local file system.

In Beta 3, we have added support for Secure Sockets Layer (SSL) protocol by integrating an SSL toolkit from IAIK. Now, ORBASEC SL2 can give you Kerberos, SSL, or both. Using a common common credentials model, such as the one in CORBA Security Level 2, makes this an easy switch, should you need to support both security protocols.

What is ORBACUS?

ORBASEC SL2 is implemented on top of ORBACUS 3.1.2 for Java, the Object Request Broker from Object Oriented Concepts, Inc. The implementation of ORBACUS allowed the introduction of ORBASEC SL2 through the ORBACUS Open Communications Interface (OCI). This particular feature of ORBACUS is beneficial because it provides the capability for "plug-able" transport mechanisms to be placed underneath the ORB request protocol (GIOP). Therefore, ORBASEC SL2 is placed on top of ORBACUS without modification to ORBACUS to give you the capability of authentication and secure encrypted communication.

What is CORBA Security Level 2?

Security Level 2 is the term used by the CORBA Security Services Specification[4] that gives a certain level of functionality to the application programmer in the form of an API. Its basic features are:

- Security Current Object
- Credentials Object
- PrincipalAuthenticator Object
- Various Runtime Security Policy Objects

Each of these objects can be queried and manipulated to get the desired security of communication and authentication.

What is SECIOP?

SECIOP stands for the SECure Inter-Operability Protocol. This *standard* protocol is specified in the CORBA Security Services Interoperability section. It is an Interoperable protocol that uses the GSS Token format standards for delivering authentication data and message protection data in a communications channel.

What is Security Replaceability?

Security Replaceable is a module specified in the CORBA Security Service specification. Its main capability is to standardize an interface so that different authentication and cryptography mechanisms can be “plugged” into the ORB security service and the SECIOP protocol.

Due to American and Canadian export laws, it may be necessary to weaken the cryptography module to be able to export the entire product out of the country.

Besides weakening, other encryption and authentication mechanisms may be able to be plugged into the ORB and still use the ORBASEC SL2 functionality, provided they conform to the CORBA Security Replaceable interfaces. Security Replaceable defines the interfaces that must be implemented so that you, the application pro-

grammer, or a third party vendor can build desired security and authentication modules and integrate them into the ORB security service in a standard fashion.

ORBASEC SL2 provides this functionality. An API is provided should the application programmer choose to create his own *Security Replaceable* Modules for SECIOP. See "Security Replaceable" on page 143.

What is Kerberos?

Kerberos is an authentication infrastructure developed at MIT and standarized at the Internet Engineering Task Force (IETF) organization.

ORBASEC SL2-GSSKRB distribution comes with a Security Replaceable Module supporting the GSS-Kerberos from the Massechusetts Institute of Technology (MIT). This distribution includes a Java Archive (JAR) file, **GSSKRB.jar**, and platform specific shared library files (or DLLs) that comprise the native implementation the MIT version of the GSS-API Kerberos Protocol. The ORBASEC SL2-GSSKRB distribution gives the applications the ability to interact with standard Internet RFC 1510[1] compliant Key Distribution Centers (KDC) for authentication services. The library also supplies the cryptography necessary for secure communication between ORB clients and ORB target objects.

What is SSL?

SSL is short for Secure Socket Layer v3.0. SSL is a socket level protocol standardized by Netscape, Inc. that sets up a secure connection between two network entities.

The ORBASEC SL2-SSL distribution comes with the SSL protocol utilizing an SSL toolkit from the Institute for Applied Information and Communication in Graz, Austria. This distribution includes a Java Archive (JAR) file, **SSLIAIK.jar**.

The ORBASEC SL2-SSL distribution gives the application writer the ability to write applications that communicate securely with other applications using authentication involving X.509 certificate based public key technology, such as DSA and RSA for authentication and secure communication. However, it does not (yet) interoperate with any established Public Key Infrastructure components, due to the

lack of mature standards in this area. Currently, ORBASEC SL2-SSL uses files containing PEM or DER encoded ceritificate chain and private key files. Should there be a commercial need for your organization to use a PKI please contact us at Adiron.

How is ORBASEC SL2 Licensed?

ORBASEC SL2 requires that several third party toolkits be installed and they must be obtained and licensed from those vendors.

- Sun JDK 1.1.x from JavaSoft, Inc.
- ORBACUS 3.1.x from Object Oriented Concepts, Inc.

ORBASEC SL2-GSSKRB requires that you have a Kerberos Version 5 compliant Key Distribution Center running that is accessible from or at your site. If you don't already have one, you can get one by licensing the following:

- MIT Kerberos 1.0.5, from the Athena Project at Massechusetts Institute of Technology

ORBASEC SL2-SSL requires a liscense from the following:

- iSaSiLk from IAIK, the Institute for Applied Information Processing and Communication, Graz, Univerity of Technology, Graz, Austria

Also, if you want to use any cryptographic algorithms owned by RSA, Inc. with the ORBASEC SL2-SSL distribution, you need:

1. A user or development license from RSA, Inc.
2. An on-site consulting agreement with Adiron, LLC to enable use of RSA with ORBAsec SL2.

ORBASEC SL2 is an open source distribution which requires developer licenses and runtime licenses in some cases. Please contact Adiron (`sales@adiron.com`) for your ORBASEC SL2 licensing needs.

About this Document

This document is layed out in such a way to give the application programmer a feel for installing and using ORBASEC SL2. This manual is also no replacement for a good book on CORBA or CORBA Security. Also, this manual is no substitute for a good book on security, authetication, encryption, or the Kerberos protocol in general.

Unfortunately, the only book out on CORBA security at the moment is the CORBA Security Specification itself.[4] This specification outlines a framework and gives a good background on how one might build a secure ORB, but it is by no means intended to be a users guide.

The Future of ORBASEC SL2

ORBASEC SL2 is not a fully functional Security Level 2 implementation, but it is fully compliant. A fully functional Security Level 2 implementation is not implementable as a single ORB library. It involves other components that provide management services, such as policy management, and access control based on those policies. The pieces that ORBASEC SL2 does not automatically use are:

- Required Rights Object
- Access Decision Object
- Auditing Decision Object
- Domain Access Policy Object

How and when the above objects are used is not exactly specified in Security Level 2. However, that does not preclude the application developer from implementing the interfaces and using them in their applications at appropriate times.

Using these components in an automatic fashion, i.e. unbeknownst to the application implementation, requires the notion of Request Level Interception. ORBACUS does not currently have support for Request Level Interception, nor has the OMG completed a fully operational specification for Request Level or Message Level Interceptors. However, a Request for Proposal (RFP) has been issued by the OMG to standarize interceptors. This RFP is known as the "RFP for Portable Interceptors."

Note – Other ORBs have support for request level interception, but provide insufficient or no support for message level interception. ORBacus gives us a form of message level interception based on their OCI interfaces which allow us to place the SECIOP and SSLIOP transport protocol into the ORB. This ability is crucial, as secure communication is handled at the transport layer.

There are other objects that ORBASEC SL2 does not use, or cannot use, because they are specified as part of the CORBA CORE, and ORBACUS does not implement them. These interfaces pertain to domain management, which is security related.

- Construction Policy Object
- Domain Manager Object

Adiron, LLC is creating several products built on top of ORBASEC SL2 that will provide centralized policy management. This capability will include, user and privilege management and centralized description of security policy including access control. However, this capability will invent, create, and make use of a higher paradigm for security than just CORBA security. Please visit our web site for updates on research and developments.

Requirements for ORBASEC SL2-GSSKRB

The following external software packages must be installed in order to run CORBA applications which use ORBASEC SL2-GSSKRB:

- Sun Java Development Kit (JDK), version 1.1.6 or later
- Object Oriented Concepts ORBACUS 3.1.2 for Java
- An Operational MIT Kerberos 5, version 1.0.5 compliant KDC

Requirements for ORBASEC SL2-SSL

The following external software packages must be installed in order to run CORBA applications which use ORBASEC SL2:

- Sun Microsystem's Java Development Kit (JDK), version 1.1.6 or later
- Object Oriented Concepts ORBACUS 3.1.2 for Java

- IAIK's iSaSiLk 2.0 toolkit and cryptographic libraries

Getting Help

There is help in the form of email to `support@adiron.com`. Also, we have set up a mailing list. To subscribe to the mailing list, send a message to `major-domo@adiron.com` (not `sl2@adiron.com`) with

```
subscribe sl2
```

in the body (e.g. not the `Subject:` field) of your message. To unsubscribe, use

```
unsubscribe sl2
```

in the body of the message. To send a message to the list, mail to `sl2@adiron.com` (not `majordomo@adiron.com`). You must subscribe to the list before you may publish to it.

CHAPTER 2

Getting Started

Getting Started

The ORBASEC SL2 distribution contains a modification of the ORBACUS "Hello World" application, which is a simple distributed application based on an introductory programming example. This application is identical to the ORBACUS "Hello World" application, except for the classes **Server** and **Client**, which have been modified to use ORBASEC SL2 security features. The source files `Server.java` and `Client.java` have been modified, accordingly.

In this chapter, we will walk through the provided source code for the **Server** and **Client** classes, discussing the security features in the ORBASEC SL2 implementation. Readers should be familiar with the ORBACUS implementation as described in the "Getting Started" chapter of the ORBACUS user manual [6].

To run the ORBASEC SL2 application in this chapter, you should have

1. The files **SL2.jar** and **GSSKRB.jar** in your **CLASSPATH** along with **OB.jar**;
2. A running Kerberos V5 KDC;
3. A valid user principal for the **Client**, such as "`user@REALM`";
4. A password for the user principal, known to the KDC;

5. *Either i)* a valid Kerberos service principal for the **Server**, such as `"host/machine.address.com@REALM"`, together with permission to read a *keytab* file in which the service principal resides, *or ii)* valid user principla for the **Server**, together with that principal's password. The demo supplied with the distribution uses the latter user-principal/password to demonstrate the use of memory keytabs in ORBASEC SL2.

Note – If you use the service-principla/keytab technique for authenticating a Server, your Kerberos Administrator may have to create a user and service principal for you, in addition to the *keytab* file which holds the service principal. Contact your Kerberos Administrator for assistance, if necessary.

Adiron's Test Kerberos Key Distribution Center (KDC)

If you use the supplied Kerberos configuration file, `orbasec_krb5.config`, this directs your Kerberos configuration to the KDC on line at Adiron. Using given principals, passwords, and keytab files, the demonstration programs should run out-of-box. This KDC is for your use in evaluation of ORBASEC SL2-GSSKRB. It will remain on-line; however, Adiron can make no guarantees that it will remain on-line indefinitely, that it will remain in the same location, or that the principal's stored in it will last forever. Adiron will make notices on its SL2 mailing list (sl2@adiron.com) if the configuration of the KDC should change.

Overview

Implementing an ORBASEC SL2 **Server** and **Client** is done in Java in much the same way as in ORBACUS, except:

- ORBASEC SL2 must be initialized in both the **Client** and **Server**; and
- The **Client** and **Server** must authenticate themselves to the Kerberos V5 KDC through the **PrincipalAuthenticator**, an object implemented by ORBASEC SL2.

Once the **Server** and **Client** have been authenticated, the resulting **Credentials** object may be modified to reflect security features supported and required by the underlying security mechanisms. In addition, security policies may be established on the **Client** to reflect application-specific policies that will be enforced by the Security Level 2 implementation.

This chapter provides an example demonstrating ORBAsec SL2 initialization and authentication through the **PrincipalAuthenticator**, together with a tutorial explaining how to modify credentials and create application-specific policies.

The IDL code

The IDL code is the same as that in the ORBacus example.

```
// IDL
interface Hello
{
    void hello();
}
```

Implementing the Example in Java

Just as in the ORBacus example, we must translate the IDL code to Java using the ORBacus IDL-to-Java compiler:

```
jidl --package hello Hello.idl
```

See the ORBacus documentation for details about the `jidl` command.

Implementing the Server

The Java implementation of the **Hello** servant is not exactly the same as that in the ORBacus example. We have modified it to print out the credentials of the client that is making the invocation on the **hello** operation.

```
// Java
package hello;
import org.omg.CORBA.*;
import org.omg.Security.*;
import org.omg.SecurityLevel2.*;

public class Hello_impl extends _HelloImplBase
{
```

```
  public ORB orb;
  public void hello()
  {
    try {
      Current current = CurrentHelper.narrow(
                      orb.resolve_initial_references(
                             "SecurityCurrent"));
      ReceivedCredentials c = current.received_credentials();
      orbasec.corba.CredUtil.dumpCredentials(System.out,c);
    } catch (Exception e) {
       e.printStackTrace(System.out);
    }
  }
}
```

The utility class **orbasec.corba.CredUtil** uses the standard CORBA Security Level 2 interfaces to display the **Credentials** object in a human readable form.

As in the ORBACUS implementation, we write a class containing a **main** method which starts up the **Hello** servant. Unlike ordinary CORBA applications, however, we must initialize ORBAsec SL2 via the **init_with_boa** static initializer on the **orbasec.SL2** class. Calling this method automatically initializes the ORB and BOA using the commnad-line options in the args parameters, together with any user-supplies **java.util.Properties**. Once SL2 is initialized, we may retrieve the ORB and BOA via the **orb** and **boa** accessors, respectively.

```
// Java
import org.omg.CORBA.*;
import java.util.Properties;

public void main(String[] args)
{
  // ORB, BOA, and SL2 initialization
  java.util.Properties properties =
                              new java.util.Properties();
  orbasec.SL2.init_with_boa( args, properties );
  ORB orb = orbasec.SL2.orb();
  BOA boa = orbasec.SL2.boa();
  ...
}
```

Once ORBASEC SL2 is initialized, we may then ask the ORB for a reference to the **SecurityLevel2::Current** object, from which we will obtain most of the security-related functionality for the **Server**:

```
// Get SecurityLevel2::Current from ORB
org.omg.CORBA.Object obj =
  orb.resolve_initial_references("SecurityCurrent");
org.omg.SecurityLevel2.Current current =
  org.omg.SecurityLevel2.CurrentHelper.narrow(obj);
...
```

Note – The **SecurityLevel2::Current** object is only available on the ORB after ORBASEC SL2 has been initialized.

With a reference to the **SecurityLevel2::Current**, we can obtain the **PrincipalAuthenticator**, the **SecurityLevel2** object we use to initialize the **Server**'s credentials. We initialize the credentials via the **PrincipalAuthenticator**'s **authenticate** method:

```
// Authenticate using PrincipalAuthenticator
org.omg.SecurityLevel2.PrincipalAuthenticator pa;
pa = current.principal_authenticator();

int    method = 0;
String mechanism = "Kerberos";
byte   security_name[] =
          orbasec.corba.Opaque.encodeKerberosName(
               "homer@MYREALM.COM"
          ).getEncoding();
byte   auth_data[] =
         ("config=FILE:orbasec_krb5.config\n" +
          "delegation=false\n" +
          "cache_name=MEMORY:0\n" +
          "enable_server=true\n" +
          "password=\"mypassword\"\n" +
          "keytab=MEMORY:0\n"
         ).getBytes();
org.omg.Security.SecAttribute privileges[] =
          new org.omg.Security.SecAttribute[0];
org.omg.SecurityLevel2.CredentialsHolder creds_holder =
          new org.omg.SecurityLevel2.CredentialsHolder();
org.omg.Security.OpaqueHolder
          continuation_data =
```

```
                    new org.omg.Security.OpaqueHolder(),
          auth_specific_data =
                    new org.omg.Security.OpaqueHolder();

pa.authenticate(
  method,
  mechanism,
  security_name,
  auth_data,
  privileges,
  creds_holder,
  continuation_data,
  auth_specific_data
);
```

The **method** parameter specifies the authentication method with which to authenticate the principal. The OMG has not specified values for this parameter, so we supply 0 (the default) as a value.

The **mechanism** parameter specifies the mechanism with which to authenticate the principal (in this case, we use the `Kerberos` mechanism).

The **security_name** parameter indicates the principal name to be recognized by the specified security mechanism. In this case, we provide a valid Kerberos 5 principal name (`"homer@MYREALM.COM"`). For ORBAsec SL2, the name must be encoded into a proper Opaque value that ORBAsec SL2 will understand. A special utility class for these encodings is **orbasec.corba.Opaque**. Please see the chapter on " Security Opaque Encodings" on page 185 for details.

Note – You may need to ask your Kerberos Administrator to create a valid principal for you.

The **auth_data** parameter in the **authenticate** method is a string converted to a byte array containing properties that are used for the GSS-Kerberos Security Mechanism. It is *essential* that each property be separated with the newline (`'\n'`) delimiter.

The above properties specify that:

- `orbasec_krb5.conf` is the configuration file that states where the KDC resides;
- This principal should have no capacity for delegation;

- The principal's credentials should be stored in a memory credentials cache indicated by `MEMORY:0`;
- That the authenticated principal is a server.
- The principal is authenticated with the password `"mypassword"`.
- The principal's key is stored in a memory type *keytab*, `MEMORY:0`.

Note – All of the definable GSS-Kerberos properties and their meanings are given in ["Authentication Data" on page 82], and the exact values of these properties will vary according to your Kerberos 5 configuration.

The ORBASEC SL2 implementation of GSS-Kerberos imposes the restriction that all server applications must have the *enable_server* property in the **auth_data** parameter set to true. Furthermore, if the **auth_data** parameter contains a *keytab* property, then the principal's key is assumed to be stored in this file, and no **password** property is needed. If the *keytab* specifies a file (i.e. prefixed by `FILE:`), then the named file, which contains the designated principal's key, must be readable by the owner of the process running the **Server**. The default *keytab* file (usually `/etc/krb5.keytab`) is read-only by the super-user, since it holds the keys to the principal names of standard *kerberized* services, such as FTP, TELNET, and LOGIN. If you need the keys in a file, you should ask your Kerberos administrator to create a special *keytab* file containing the principals you may use for the example **Server**. A keytab file is needed if the principal has a "randomized" key, which means that it does not have a password. If the principal has a password, then a memory keytab can be specified which does not expose the principal's encryption key to a file system.

The **privileges** parameter specifies privileges that must be authenticated through the security mechanism. The GSS-Kerberos security mechanism provides no support for such privileges, so we pass an empty **Security::SecAttribute** list.

Once the server is authenticated, the **Credentials** object is returned in the **CredentialsHolder** structure; they are also stored on the **SecurityLevel2::Current** object's **own_credentials** list attributefor easy access from other parts of the program.

The **continuation_data** and **auth_specific_data** output parameters are used with security mechanisms that support multi-step authentication protocols. The GSS-Kerberos security mechanism only supports single-step authentication, so the output parameter values are ignored.

The remainder of the program is the same as it is in ORBACUS. An instance of the **Hello_impl** class is created, the IOR for that servant is written to a file, and the **BOA** starts servicing requests from clients via the **impl_is_ready** method. (See the ORBACUS documentation for sample code).

Note – The **Server** must authenticate a principal and obtain a credentials object before the IOR is advertised to clients. This procedure is necessary because the IOR contains mechanism-specific data that a client will need to use in order to communicate securely with the server. If the **Server** has not authenticated a principal, no security information will get advertised in the Hello object's IOR, and an exception will be raised upon making a request.

Implementing the Client

The **Client** implementation is much the same as it is in ORBACUS, except that, like the **Server** implementation, the **Client** must be authenticated through the **PrincipalAuthenticator**'s **authenticate** method. As before, a reference to the **PrincipalAuthenticator** is obtained through the **SecurityLevel2::Current**

```
// Java
import org.omg.CORBA.*;
import java.util.Properties;

public void main(String[] args)
{
  // ORB, BOA, and SL2 initialization
  Properties properties = new Properties();
  orbasec.SL2.init( args, new Properties() );
  ORB orb = orbasec.SL2.orb();

  // Get SecurityLevel2::Current from ORB
  org.omg.CORBA.Object obj =
    orb.resolve_initial_references("SecurityCurrent");
  org.omg.SecurityLevel2.Current current =
    org.omg.SecurityLevel2.CurrentHelper.narrow(obj);
```

Note – In general, there need not be a **BOA** to accept requests for client applications, since client applications are not typically CORBA objects.

Client authentication is similar to that of the **Server**:

```
current = // get reference to Current

// Authenticate using PrincipalAuthenticator
org.omg.SecurityLevel2.PrincipalAuthenticator pa;
pa = current.principal_authenticator();

int     method = 0;
String  mechanism = "Kerberos";
byte    security_name[] =
            orbasec.corba.Opaque.encodeKerberosName(
                 "bart@MYREALM.COM"
            ).getEncoding();
byte    auth_data[] =
            ("config=FILE:orbasec_krb5.config\n" +
             "delegation=false\n" +
             "cache_name=MEMORY:0\n" +
             "password=\"mypassword\"\n"
            ).getBytes();
org.omg.Security.SecAttribute privileges[] =
            new org.omg.Security.SecAttribute[0];
org.omg.SecurityLevel2.CredentialsHolder creds_holder =
            new org.omg.SecurityLevel2.CredentialsHolder();
org.omg.Security.OpaqueHolder
            continuation_data =
                   new org.omg.Security.OpaqueHolder(),
            auth_specific_data =
                   new org.omg.Security.OpaqueHolder();

pa.authenticate(
  method,
  mechanism,
  security_name,
  auth_data,
  privileges,
  creds_holder,
  continuation_data,
  auth_specific_data
);
```

The **method** parameter specifies the authentication method with which to authenticate the principal. The OMG has not specified values for this parameter, so we supply 0 (the default) as a value.

The **mechanism** parameter specifies the mechanism with which to authenticate the principal (in this case, we use the Kerberos mechanism).

The **security_name** parameter indicates the principal name to be recognized by the specified security mechanism. In this case, we provide a valid Kerberos 5 principal name (`"bart@MYREALM.COM"`). Like the server, this must be in the special Opaque encoding of a Kerberos Name. Please see chapter on "Opaque Encodings" on page 185 for further details.

Note – You may need to ask your Kerberos Administrator to create a valid principal for you. You will need a valid password for this principal, as well.

The **auth_data** parameter in the **authenticate** method is a byte array containing properties that are used in the GSS-Kerberos Security Mechanism. Please note that it is *essential* that each property be separated with the newline (`'\n'`) delimiter.

The above properties specify that:

- `orbasec_krb5.conf` is the configuration file that states where the KDC resides;
- This principal should have no capacity for delegation;
- The principal's credentials should be stored in a memory credentials cache indicated by `MEMORY:0`;
- The principal is authenticated with the password `"mypassword"`.

Note – All of the definable GSS-Kerberos properties and their meanings are given in ["Authentication Data" on page 82], and the exact values of these properties will vary according to your Kerberos 5 configuration.

The ORBASEC SL2 implementation of GSS-Kerberos imposes the convention that if the **auth_data** parameter does *not* contain a *keytab* property (or if it is empty), then the principal's credentials must be obtained in one of two ways: if the **auth_data** parameter contains a password property, then the principal should be authenticated using the designated password; if, on the other hand, the **auth_data** parameter does not contain a **password** property (or if it is empty), then the Kerberos client should already have been authenticated externally (e.g., via the Kerberos *kinit* program). In this case, a designated cache file should already contain the principal's Kerberos credentials. To designate a cache file, the **cache_name** property should have the form "`FILE:`<*filename*>". If the **cache_name** property is empty, then the default cache is used. This cache will be a file named by "`/tmp/`

`krb5cc_<uid>`" on Unix systems where *uid* is the user number of the user that is logged on. See the ORBASEC property "orbasec.kerberos_session" on page 51 for how to automatically initialize Kerberos session credentials during SL2 initialization.

If you attempt to use the default credentials cache file without a password, the Kerberos name supplied in the **security_name** parameter must match those in the credentials cache file or a GSS Exception will be thrown. Alternatively, you may set the **security_name** parameter to `null` or `new byte[0]` to automatically use the name in the credentials cache file.

The **privileges** parameter specifies privileges that must be authenticated through the security mechanism. The GSS-Kerberos security mechanism provides no support for such privileges, so we pass an empty **Security::SecAttribute** list.

Once the server principal is authenticated, a **Credentials** object is returned in the **CredentialsHolder** structure; the **Credentials** object is also stored on the **SecurityLevel2::Current::own_credentials** attribute for easy access from other parts of the program.

The **continuation_data** and **auth_specific_data** output parameters are used with security mechanisms that support multi-step authentication protocols. The GSS-Kerberos security mechanism only supports single-step authentication, so the output parameter values are ignored.

The remainder of the program is the same as it is in ORBACUS. A reference to the **Hello** object is obtained from the published IOR, and the program enters a loop calling the **hello** method of the referenced object. (See the ORBACUS documentation for sample code).

The **Client** should be authenticated via the **PrincipalAuthenticator** after the ORB and ORBASEC SL2 have been initialized and before making any requests on the **Hello** object.

Compiling the Demo

The procedure for compiling the demo is fairly straight forward, should you be familiar with make files. From within the `sl2/demo/krb-hello` directory, run the command:

```
make
```

You need to make sure that you have the ORBACUS jidl command in your execution path, and you must have OB.jar, SL2.jar, GSSKRB.jar in your Java CLASSPATH.

You should see the following output:

```
mkdir classes
mkdir generated
jidl --tie --package hello --output-dir generated Hello.idl
CLASSPATH=.:./classes:$CLASSPATH \
javac -deprecation -d classes \
generated/hello/*.java
```

Running the Demo

Running the demo involves starting the Server and then starting the Client. The Server must be started first, because it writes out the IOR of the Hello object to a file called Hello.ref.

Running the Server

To run the Server, type:

```
java hello.Server
```

You should see the following output:

```
Own Credentials:
Credentials:
    credential_type              = SecOwnCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  =
[NoProtection,Integrity,Confidentiality,DetectReplay,EstablishTrustInTarget,Est
ablishTrustInClient,NoDelegation,SimpleDelegation]
    accepting_options_required   = [EstablishTrustInClient]
    invocation_options_supported =
[NoProtection,Integrity,Confidentiality,DetectReplay,EstablishTrustInTarget,Est
ablishTrustInClient,NoDelgation]
    invocation_options_required  = [EstablishTrustInClient]
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,2,”Adiron”,”30514”)
     SecAttribute(41244,1,1,”Adiron”,”128.230.99.3”)
     SecAttribute(0,0,0,””,”Kerberos_MIT”)
```

```
     SecAttribute(0,1,2,”krbtgt/MYREALM.COM@MYREALM.COM”,”bart@MYREALM.COM”)
Hello Server is Ready.
```

The Server authenticates its principal and then displays its credentials. You may want observe the **accepting_options_supported** and **accepting_options_required** attributes as these will change if you modify the demo according to the following sections.

The security attributes listed at the bottom of the listing contain the attributes of the Credentials that have been dumped to the screen. Attributes are typed by 3 numbers, the Family Definer, and Family, and then the type. The last one listed is the AccessId attribute, which belongs to the CORBA (0) Family (1) and is Type (2).

Adiron has its own families of security attributes, Family Definer of (41244, i.e. 0xA11C).

Family 0 Type 1 is the security mechanism.

Family 1 pertains to network addresses. Family 1 Type 1 names the local IP host address. Family 1 Type 2 names the local IP port number. Family 1 Type 3 names the remote IP host address, and Family 1 Type 4 names the remote IP port number.

Note – These Adiron IP attributes will have different values than printed here when you run the programs.

Running the Client

To run the Client, type:

```
java hello.Client
```

You should see the following output:

```
Own Credentials:
Credentials:
    credential_type              = SecOwnCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported  =
[NoProtection,Integrity,Confidentiality,DetectReplay,EstablishTrustInTarget,Est
ablishTrustInClient,NoDelgation]
    invocation_options_required  = [EstablishTrustInClient]
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,2,”Adiron”,”30515”)
```

```
     SecAttribute(41244,1,1,”Adiron”,”128.230.99.3”)
     SecAttribute(0,0,0,””,”Kerberos_MIT”)
     SecAttribute(0,1,2,”krbtgt/MYREALM.COM@MYREALM.COM”, ”marge@MYREALM.COM”)

Getting Hello Reference.

Hello’s Credentials:
Credentials:
    credential_type              = SecOwnCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported = []
    invocation_options_required  = []
    invocation_options_used =
[NoProtection,Integrity,Confidentiality,DetectReplay,EstablishTrustInTarget,Est
ablishTrustInClient,NoDelgation]
    delegation_mode              = SecDelModeNoDelegation
    delegation_state             = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,”Adiron”,”30514”)
     SecAttribute(41244,1,3,”Adiron”,”128.230.99.3”)
     SecAttribute(41244,1,2,”Adiron”,”30515”)
     SecAttribute(41244,1,1,”Adiron”,”128.230.99.3”)
     SecAttribute(0,0,0,””,”Kerberos_MIT”)
     SecAttribute(0,1,2,”krbtgt/MYREALM.COM@MYREALM.COM”, ”bart@MYREALM.COM”)
Enter ‘h’ for hello or ‘x’ for exit:
>
```

You will notice in contrast to the Server principal’s credentials that since the Client’s principal was authenticated without naming a *keytab*. It is a pure client. The **accepting_options_supported** field is empty, as these credentials cannot be used to accept secure associations.

To continue with the demo, type ‘h’ as requested and you should see the following output on the Server’s side:

```
Credentials:
    credential_type              = SecReceivedCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported = []
    invocation_options_required  = []
    association_options_used     =
[Integrity,Confidentiality,DetectReplay,EstablishTrustInClient,NoDelgation]
    delegation_mode              = SecDelModeNoDelegation
    delegation_state             = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,”Adiron”,”30515”)
     SecAttribute(41244,1,3,”Adiron”,”128.230.99.3”)
     SecAttribute(41244,1,2,”Adiron”,”30514”)
     SecAttribute(41244,1,1,”Adiron”,”128.230.99.3”)
     SecAttribute(0,0,0,””,”Kerberos_MIT”)
     SecAttribute(0,1,2,”krbtgt/MYREALM.COM@MYREALM.COM”, ”marge@MYREALM.COM”)
```

You will notice that these are “received” credentials printed out by the Hello object implementation, **Hello_impl**. Since this invocation was made without delegation, all accepting and invocation options are empty, as these credentials may not be used to accept secure associations or to initiate them (used to make invocations). Since these are **ReceivedCredentials**, there are extra attributes, such as **association_options_used**, **delegation_mode**, **delegation_state**. The **delegation_mode** indicates the delegation ability of these credentials. Here, it is no delegation. The **delegation_state** attribute indicates that the client is the principal that made the invocation. The **association_options_used** are the association options that were used in the negotiated secure association with the client. You will notice that **Integrity** and **Confidentiality** were both used, and **EstablishTrustInClient**. However, you will notice that **EstablishTrustInTarget** is absent, indicating that the Server did not authenticate itself to the Client, i.e. there was no mutual authentication. This absence of mutual authentication is the result of the Server and/or Client not requiring trust in the target to be established.

Modifying the Server

The above example demonstrates minimal and default capabilities of the ORBASEC SL2-GSSKRB implementation of Security Level 2. However, ORBASEC SL2 provides functionality through the Security Level 2 interfaces for using most of the security features provided. In this section, we provide a few modifications to the “Hello World” application to illustrate this functionality.

Server Accepting Options

After the **Server** is authenticated through the **PrincipalAuthenticator**, the **Credentials** for the **Hello** servant includes information about the servant’s “accepting options”, security features it will support or require when a **Client** makes an invocation. These features are published in the object’s IOR so that clients making requests can communicate securely with servants without having to go through a complicated and costly protocol to establish secure communication.

Each **Credentials** object represents a security mechanism component that is advertised in a object’s IOR.

The accepting options are defined in the CORBA Security Specification to be: *NoProtection*, *Integrity*, *Confidentiality*, *DetectReplay*, *DetectMisordering*, *Estab-*

lishTrustInTarget, and *EstablishTrustInClient*. Each option is specified to be supported or required, with the restriction that no feature can be required if it is not supported. Table 1 on page 36 shows the default values for these options in the ORBASEC SL2 implementation of GSS-Kerberos.

Feature	Supported	Required
NoProtection	yes	no
Integrity	yes	no
Confidentiality	yes	no
DetectReplay	yes	no
DetectMisordering	no	no
EstablishTrustInTarget	yes	no
EstablishTrustInClient	yes	yes
NoDelegation	yes	no
SimpleDelegation	yes	no
CompositeDelegation	no	no

TABLE 1. GSS-Kerberos Default Server Accepting Options

Accepting options are stored in the Server's own **Credentials** object, which is obtained after authentication using the **PrincipalAuthenticator**. We can change the accepting options by using the **accepting_options_required** and **accepting_options_supported** attribute accessor methods of the **Credentials** object to manipulate the options that are required and the options that are supported, respectively.

In the example below we require that a client must use mutual authentication by turning on the **EstablishTrustInTarget** bit. (We say "mutual authentication," because **EstablishTrustInClient** is always required and is already set.) Setting this option has the effect of telling the client not to send any messages to the target until it has verified the server's identity.

Accepting Options are represented by constants of the **Security::AssociationOptions** type, which are bit positions. Therefore, changing them requires the use of bitwise operators, "&", "|", "~".

```
// Authenticate using PrincipalAuthenticator
...
```

```
// Modify Accepting Options
org.omg.SecurityLevel2.Credentials[] credlist =
                                    current.own_credentials();
// Get our Kerberos credentials from the own credentials list
org.omg.SecurityLevel2.Credentials creds = credlist[0];
creds.accepting_options_required( (short)
          (creds.accepting_options_required() |
            org.omg.Security.EstablishTrustInTarget.value));
```

We can turn support of **NoProtection** off as follows:

```
creds.accepting_options_supported( (short)
           (creds.accepting_options_supported() &
             ~org.omg.Security.NoProtection.value));
```

Any client that gets the published IOR for this **Server** will know that the **Server** requires that the client establish trust in the server in order to make a connection, and furthermore that the **Server** does not support unprotected messages.

The **Server** accepting options should be modified before publishing the IOR to prospective clients (i.e. by using the `object_to_string` method or returning an object reference) since the accepting options information is recorded in the IOR. Clients use the IOR to make decisions about the security features to use based on the **Server**'s accepting options, together with the client's invocation policies (see below).

If you recompile and rerun the demo, then you will notice two things. First the "own" credentials will print out as follows:

```
Own Credentials:
Credentials:
    credential_type              = SecOwnCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  =
[NoProtection,Integrity,Confidentiality,DetectReplay,EstablishTrustInTarget,Est
ablishTrustInClient,NoDelegation,SimpleDelegation]
    accepting_options_required   =
[EstablishTrustInTarget,EstablishTrustInClient]
    invocation_options_supported =
[NoProtection,Integrity,Confidentiality,DetectReplay,EstablishTrustInTarget,Est
ablishTrustInClient,NoDelgation]
    invocation_options_required  = [EstablishTrustInClient]
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,2,"Adiron","30514")
     SecAttribute(41244,1,1,"Adiron","128.230.99.3")
     SecAttribute(0,0,0,"","Kerberos_MIT")
     SecAttribute(0,1,2,"krbtgt/MYREALM.COM@MYREALM.COM","bart@MYREALM.COM")

Hello Server is Ready.
```

You will notice that **EstablishTrustInTarget** is now in the **accepting_options_required** attribute. After typing ‘h’ on the Client the following will be printed out:

```
Credentials:
    credential_type              = SecReceivedCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported = []
    invocation_options_required  = []
    association_options_used     =
[Integrity,Confidentiality,DetectReplay,EstablishTrustInTargett,EstablishTrustI
nClient,NoDelgation]
    delegation_mode              = SecDelModeNoDelegation
    delegation_state             = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,”Adiron”,”30514”)
     SecAttribute(41244,1,3,”Adiron”,”128.230.99.3”)
     SecAttribute(41244,1,2,”Adiron”,”30515”)
     SecAttribute(41244,1,1,”Adiron”,”128.230.99.3”)
     SecAttribute(0,0,0,””,”Kerberos_MIT”)
     SecAttribute(0,1,2,”krbtgt/MYREALM.COM@MYREALM.COM”, ”marge@MYREALM.COM”)
```

You will notice that mutual authentication was established with the client, indicated by the **EstablishTrustInTarget** in the **association_options_used** attribute.

Modifying the Client

The Client can be modified as well to use policies to direct the characteristics of invocations.

Invocation Policies

From the security point of view, when a client makes an invocation on a method of a remote object, several decisions need to be made about the communication that is to take place between client and server. These decisions include, but by no means are limited to:

- What credentials will be used by the client to make a secure association (i.e., whether to use the client's "own" credentials or credentials it might have obtained as a result of an invocation on it, its "received" credentials);
- What security mechanisms (e.g., GSS-Kerberos, SSL, etc.) will be used to form a secure association;

- Whether messages sent between the client and server will be encrypted, have a facility for integrity, neither, or both;
- Whether the client will authenticate itself to the server, whether the server will authenticate itself to the client, neither, or both; and
- Whether the client may delegate the server to make remote invocations on other objects on the client's behalf.

The server has some say in these decisions; it publishes (through its IOR) what security features it requires or supports, and we have seen above how to modify these options. In addition, however, the Client has some say in these decisions through the use of **CORBA::Policy** objects. These "invocation policies" specify how the client should make attempt to a secure association with a server, in the absence of knowing anything about what the server supports or requires. Given a collection of invocation policies, together with information 1) about what a server supports and requires (through the publicized IOR), and 2) and what the client's credentials are, ORBASEC SL2 can then make decsions about whether a secure association is possible, and if so, what security features will be used to make the association.

The Security Level 2 interfaces define five kinds of **CORBA::Policy** objects that clients can adopt. They are *Invocation Credentials Policy, Mechanism Policy, QOP Policy, Delegation Directive Policy, and Establish Trust Policy*. The precise definitions of these **CORBA::Policy** objects is beyond the scope of this tutorial (see “Policies” on page 125 for a more complete description), but a few remarks can be made at this preliminary stage. First, ORBASEC SL2 includes default behaviors for these policies, so that if none are explicity set in the client, predictable behavior can be expected. These defaults are summarized in Table “Initial Default Policies on the Current” on page 40.

Another important point is that in the CORBA object model, there are effectively two ways to set the invocation policies from a client to a server. The first is to use the **orbasec.SecLev2::Current::set_overrides** method, with a list of Policy objects as the argument. This has the effect of setting the "default" or "environment" policies, so that any request inititiated after that point will use those policies (or the defaults, if a policy of one of the above 5 types was not specified).

Since clients may have many references to remove objects, however, this method for setting policies can be cumbersome. So in addition CORBA supplies the **_set_policy_overrides** psuedo operation, which is a operation supplied on a **CORBA::Object**. The intention here is to designate specific invocation policies

on an object reference, so that the defaults do not have to be changed every time a new reference is obtained..

TABLE 2. Initial Default Policies on the Current

Policy	Default
Invocation Credentials	Use the Received and Own Credentials that support invocation.
Mechanism	Use mechanisms of the Credentials in the Invocation Credentials policy.
QOP	QOP required by the first credentials in the Invocation Credentials Policy
Delegation Directive	No Delegation
Establish Trust	Trust in Client, if required or supported by the first credentials in the Invocation Credentials Policy. Trust in Target, if required or supported by the first credentials in the Invocation Credentials Policy.

For more information about policies in general, see the CORBA Specification [2]. For more details on these specific policies see Chapter "Policies" on page 125.

Changing Policies on Current

Both default and object specific policies are configurable in ORBASEC SL2. We modify the default policies on the **orbasec.SecLev2.Current** by creating an array of **Policy** objects using **orbasec.SL2** factory methods and creating the new policies:

```
// Modify Invocation Policies
org.omg.CORBA.Policy[] policies =
                  new org.omg.CORBA.Policy[2];
current = // get the orbasec.SecLev2.Current object ...
policies[0] = orbasec.SL2.create_qop_policy(
                  org.omg.Security.QOP.SecQOPIntegrity);

// Set overrides on Current PolicyManager
current.set_overrides(policies,
                        org.omg.CORBA.ADD_OVERRIDE.value);
```

Note – The **orbasec.SecLeve2.Current** object is an ORBAsec SL2 extension of **org.omg.SecurityLevel2.Current** that has support for setting policies on the current thread. Standardization of this feature is pending at the OMG.

These policies specify to use “integrity” only (not confidentiality and integrity together). After placement on the **Current**, they will now be used by any remote object reference which does not specifically override these policies (see below).

Recompile and run the Client. After hitting ‘h’ you will see the following output from the Server:

```
Credentials:
    credential_type              = SecReceivedCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported = []
    invocation_options_required  = []
    association_options_used     =
[Integrity,DetectReplay,EstablishTrustInTargett,EstablishTrustInClient,NoDelgat
ion]
    delegation_mode              = SecDelModeNoDelegation
    delegation_state             = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,”Adiron”,”30515”)
     SecAttribute(41244,1,3,”Adiron”,”128.230.99.3”)
     SecAttribute(41244,1,2,”Adiron”,”30514”)
     SecAttribute(41244,1,1,”Adiron”,”128.230.99.3”)
     SecAttribute(0,0,0,””,”Kerberos_MIT”)
     SecAttribute(0,1,2,”krbtgt/MYREALM.COM@MYREALM.COM”,
”hello_demo_client@MYREALM.COM”)
```

You will notice that **Confidentiality** is not in the **association_options_used**, but **Integrity** is.

Policies should be chosen that are consistent with the security features advertised in a target objects’s IOR. One cannot, for example, use a policy which states to use no protection if it is not supported by the **Server**. By the same token, policies should be specified if they are advertised to be required in the IOR; if the **Server** requires trust in a client, for example, the policy should reflect this requirement. Otherwise, a **CORBA::NO_RESOURCES** exception may be raised with the reason of “No matching credentials available”.

Since our new Server has shut off support for **NoProtection**, change the **SecQOPIntegrity** policy to one of **SecQOPNoProtection** and recompile and rerun the client. You will get the following output from the Client.

```
Getting Hello Reference
No Matching credentials available
  Policy mechanisms: Kerberos_MIT
  IOR mechanisms: Kerberos
 Credential: Kerberos_MIT[<some address>
  Target does not support selected options 0x41 target
supports 0x1ee
  <stack trace>
```

You will not even get to the Client's prompt because in creating the hello reference, a valid security context had to have the ability to be created. In this case, due to the lack of commonality between the supported features of the target, the client side policies, and the client side credentials, no secure association could be established.

Change the QOP Policy set on Current back to one of **SecQOPIntegrity** and proceed to the next section.

Changing Policies on Object References

The second way to override policies is to associate a set of policies with a specific object reference. This is done by creating an array of **Policy** objects and registering them with the object using the **_set_policy_overrides** operation on the object reference:

```
hello = // Obtain Object reference somehow...
policies = new org.omg.CORBA.Policy[1];

// QOP Policy: use Integ and Conf!
policies[0] =
orbasec.SL2.create_qop_policy(
     org.omg.Security.QOP.SecQOPIntegrityAndConfidentiality );

// Set the policy on the hello_2 Object reference
hello_2 = HelloHelper.narrow(
             hello._set_policy_overrides(
                 policies,
                   org.omg.CORBA.ADD_OVERRIDE.value));
```

The above code turns on integrity and confidentiality when an invocation is made through the new **hello_2** reference.

The above overrides do not effect policies associated with the reference on which **_set_policy_overrides** was called (viz., **hello**); invocations through the **hello** reference, for example, will still use the default **QOPPolicy** on **Current**, i.e. integrity and confidentiality. Our demonstration program is set up to make two invocations when the 'h' is hit, one with the **hello** object reference and the next one is with the **hello_2** object reference. The output from the Server is as follows:

```
Credentials:
    credential_type              = SecReceivedCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported = []
    invocation_options_required  = []
    association_options_used     =
[Integrity,DetectReplay,EstablishTrustInTargett,EstablishTrustInClient,NoDelgat
ion]
    delegation_mode              = SecDelModeNoDelegation
    delegation_state             = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,"Adiron","30515")
     SecAttribute(41244,1,3,"Adiron","128.230.99.3")
     SecAttribute(41244,1,2,"Adiron","30514")
     SecAttribute(41244,1,1,"Adiron","128.230.99.3")
     SecAttribute(0,0,0,"","Kerberos_MIT")
     SecAttribute(0,1,2,"krbtgt/MYREALM.COM@MYREALM.COM", "marge@MYREALM.COM")

Credentials:
    credential_type              = SecReceivedCredentials
    mechanism                    = Kerberos_MIT
    accepting_options_supported  = []
    accepting_options_required   = []
    invocation_options_supported = []
    invocation_options_required  = []
    association_options_used     =
[Integrity,Confidentiality,DetectReplay,EstablishTrustInTargett,EstablishTrustI
nClient,NoDelgation]
    delegation_mode              = SecDelModeNoDelegation
    delegation_state             = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,"Adiron","30515")
     SecAttribute(41244,1,3,"Adiron","128.230.99.3")
     SecAttribute(41244,1,2,"Adiron","30514")
     SecAttribute(41244,1,1,"Adiron","128.230.99.3")
     SecAttribute(0,0,0,"","Kerberos_MIT")
     SecAttribute(0,1,2,"krbtgt/MYREALM.COM@MYREALM.COM", "marge@MYREALM.COM")
```

You will notice the difference in the second **Credentials** object that is printed out. **Confidentiality** is on, indicating that successful encrypted communication of the second request was in effect.

Where to Go From Here

The remaining chapters provide a thorough description of the application programmer's interface to ORBASEC SL2. You should have a basic understanding of CORBA Security as detailed in [4] before proceeding. We also encourage you to work with the ORBASEC SL2 implementation and experiment with various accepting option and invocation policy combinations. Doing so will provide a hands-on familiarity with a small part of CORBA Security, particularly if you have the CORBA Security Specification within arm's reach.

There are number of source code demonstration tests in the form of directories under the `sl2/demo` directory. These tests exercise both the ORBASEC SL2-GSSKRB and ORBASEC SL2-SSL distributions, and can be used by you to experiment with CORBA Security Level 2 functionality.

CHAPTER 3

SL2 Initialization

Initializing SL2

The **orbasec.SL2** class provides a collection of static initialization methods for initializing a secure ORB. Each of these methods initializes the ORB automatically, so you should not initialize the ORB or BOA before calling any of these methods. The following sections describe the conditions under which you should use these intializers.

Standalone Server Initialization

If you are initializing a standalone server and require a BOA to dispatch requests to servants, use the **init_with_boa** method

```
static void init_with_boa(
    String              argv[],
    java.util.Properties properties );
```

This method will initialize the ORB and BOA, and then initialize the ORBAsec security infrastructure. The **argv** and **properties** arguments behave just as they do in the ORB.init method. (**Note.** We feel the properties passed to the **boa_init** method of the ORB are typically superfluous, so we have combined them into a single collection of properties.) See See “ORBAsec SL2 Configuration” on page 47. for rules governing the precedence of ORBAsec properties.

Stanalone "Pure" Client Initialization

If you are initializing a standalone "pure" client, i.e, you do *not* require a BOA to dispatch requests to servants, use the **init** method

```
static void init(
    String              argv[],
    java.util.Properties properties );
```

This method will initialize the ORB, and then initialize the ORBASEC security infrastructure. The **argv** and **properties** arguments behave just as they do in the ORB.init method with the same signature. See See "ORBAsec SL2 Configuration" on page 47. for rules governing the precedence of ORBAsec properties.

Applet Initialization

If you are initializing a Java Applet, use the **init** method

```
static void init(
    java.applet.Applet  applet,
    java.util.Properties properties );
```

This method will initialize the ORB, and then initialize the ORBASEC security infrastructure. The **applet** and **properties** arguments behave just as they do in the ORB.init method with the same signature. See See "ORBAsec SL2 Configuration" on page 47. for rules governing the precedence of ORBAsec properties.

Accessors

Once one of the above initializers is called, you may use the **orb** and **boa** accessors

```
static org.omg.CORBA.ORB orb();
static org.omg.CORBA.BOA boa();
```

to obtain a reference to the ORB (and BOA, if initialized) that was initialized in ORBASEC.

ORBASEC SL2 Configuration

ORBASEC SL2 defines a set of Java Properties that can be used to configure security protocols, mechanisms, and other features at ORBASEC SL2 initialization. These properties can be specified in one of the following ways:

- via an ORBACUS configuration file;
- via the **java.util.Properties** arguments to one of the **orbasec.SL2** initializers;
- via System Property definitions (within a Java program or from the command line, -D on most systems); or
- via command-line options

To define an ORBASEC SL2 property via a configuration file, use the ORBACUS -*ORBconfig* command-line option. See the ORBACUS manual for how to use this option and for the syntax of the ORBACUS configuration file.

Command-line options override Java System Property definitions, which in turn override properties defined in the **java.util.Properties** argument passed to an initializer, which in turn override properties defined in an ORBACUS configuration file. If no property is defined, an appropriate default is used. This behavior mirrors that of property definitions in ORBACUS.

Standard ORBASEC SL2 Properties

This section enumerates the standard ORBASEC SL2 properties likely to be used by the Application Programmer, together with their meanings and default values. See “ Adding your own Security Mechanisms” on page 55 for more ORBASEC SL2 properties.

orbasec.seciop

This property determines whether the ORBASEC SECIOP protocol should be enabled in client or server mode, or whether SECIOP should be disabled all together. If SECIOP is not enabled in server mode, any CORBA servants will not be allowed to accept SECIOP connections.

Legal Values

client	enable SECIOP in client mode

server enable SECIOP in (client and) server mode
disable disable SECIOP

Default Value
server

orbasec.seciop.host

Use this property to specify a host for SECIOP connections. If this property is not defined, ORBASEC will use the **ooc.boa.hostname** property value, or the local canonical hostname, if that property is not defined.

Legal Values
any legal host name or IP address

Default Value
none

Note – The **orbasec.seciop** property must equal **server** in order for this property to have any effect.

orbasec.seciop.port

Use this property to specify a port for SECIOP connections. If this property is not defined, ORBASEC will use the **orb.boa.port** property value, or the port chosen by the ORB, if that value is not defined or defined to be zero.

Legal Values
any port number you are permitted to open

Default Value
none

Note – The **orbasec.seciop** property must equal **server** in order for this property to have any effect. Ports under 1024 need "root" privilege.

orbasec.ssliop

This property determines whether the ORBASEC SSLIOP protocol should be enabled in client or server mode, or whether SSLIOP should be disabled all together. If SSLIOP is not enabled in server mode, any CORBA servants will not be allowed to accept SSLIOP connections.

Legal Values

client	enable SSLIOP in client mode
server	enable SSLIOP in (client and) server mode
disable	disable SSLIOP

Default Value

server

orbasec.ssliop.host

Use this property to specify a host for SSLIOP connections. If this property is not defined, ORBASEC will use the **ooc.boa.hostname** property value, or the local canonical hostname, if that property is not defined.

Legal Values

any legal host name or IP address

Default Value

none

Note – The **orbasec.ssliop** property must equal **server** in order for this property to have any effect.

orbasec.ssliop.port

Use this property to specify a port for SSLIOP connections. If this property is not defined, ORBASEC will use the **orb.boa.port** property value, or the port chosen by the ORB, if that value is not defined or defined to be zero.

Legal Values

any port number you are permitted to open

Default Value
none

Note – The **orbasec.ssliop** property must equal **server** in order for this property to have any effect. Ports under 1024 need "root" privilege.

orbasec.ssliop.exportable_only

This property states whether only exportable ecncryption cipher suites are available for selection.

U.S. Export laws stipulate the cryptographic strength used for encryption. Setting this property to **true** will limit the cipher suites to only "exportable" cipher suites.

Legal Values

true	Limit exportable only
false	No limit on cipher suites.

Default Value
true

orbasec.iiop

This property determines whether the ORBASEC IIOP protocol should be enabled in client or server mode, or whether IIOP should be disabled all together. If IIOP is not enabled in server mode, any CORBA servants will not be allowed to accept IIOP connections.

IIOP is the protocol used for general CORBA standard (insecure) communication.

Legal Values

client	enable IIOP in client mode
server	enable IIOP in (client and) server mode
disable	disable IIOP

Default Value
disable

orbasec.iiop.host

Use this property to specify a host for IIOP connections. If this property is not defined, ORBASEC will use the **ooc.boa.hostname** property value, or the local canonical hostname, if that property is not defined.

Legal Values
any legal host name or IP address

Default Value
none

Note – The **orbasec.iliop** property must equal **server** in order for this property to have any effect.

orbasec.iiop.port

Use this property to specify a port for IIOP connections. If this property is not defined, ORBASEC will use the **orb.boa.port** property value, or the port chosen by the ORB, if that value is not defined or defined to be zero.

Legal Values
any port number you are permitted to open

Default Value
none

The **orbasec.iiop** property must equal **server** in order for this property to have any effect. Ports under 1024 need "root" privilege.

orbasec.kerberos_session

Setting this property to **true** automatically creates a Kerberos **Credentials** object that is initialized with the current user's Kerberos session credentials cache (as obtained from a program such as *kinit*). On most Unix systems, (some systems have different system defaults) the credentials cache file `/tmp/krb5cc_<uid>`, where `<uid>` is the current users uid, or it is the value of the `KRB5CCACHE` environment variable. It also takes the kerberos configuration file to be `/etc/krb5.conf` or the value of the `KRB5_CONFIG` environment variable.

Note – When running with a Credentials object initialized from the Kerberos session cache, the process can only use this **Credentials** object in a client fashion. That is, the process does not associate these credentials with object references produced by the ORB.

If you need for the server publish object references with SECIOP-Kerberos credentials information, the Kerberos Credentials objects must be explictly created in application code.

Legal Values

true	use Kerberos session credentials cache
false	require explicit Kerberos authentication

Default Value

false

orbasec.anonymous_ssl

Setting this property to **true** automatically creates an anonymous SSL **Credentials** object during ORBASEC initialization. No certificate file is required for anonymous SSL credential initialization. It uses the Diffe-Hillman cipher suites.

Legal Values

true	use anonymous SSL credentials
false	require explicit SSL authentication

Default Value

false

Note – The **orbasec.ssliop** property must *not* be set to **disable** in order for this property to have any effect.

orbasec.allow_iiop

Setting this property to **true** automatically creates an IIOP **Credentials** object during ORBASEC initialization.

This credentials object must be created to enable IIOP (insecure) communication over the CORBA standard protocol.

Legal Values

true	use ORBAsec IIOP credentials
false	require explicit IIOP authentication

Default Value

false

Note – The **orbasec.iiop** property must *not* be set to **disable** in order for this property to have any effect.

ORBASEC SL2 Command-line Options

ORBASEC SL2 provides command-line options for specifying the values of ORBASEC SL2 properties at initialization. You may use these command-line arguments in conjunction with ORBACUS command-line options.

The ORBASEC SL2 command-line options provide the ability to override ORBAsec properties defined in a configuration file or via the System Property definition flag to the Java Virtual Machine (-D on most systems). Command-line usage is summarized in table 3, and the meanings of each flag is the same as that of

ORBASEC SL2 Command-line Option	ORBASEC SL2 Property
-SL2SECIOP *mode*	orbasec.seciop=*mode*
-SL2SECIOPHost *host*	orbasec.seciop.host=*host*
-SL2SECIOPPort *port*	orbasec.seciop.port=*port*
-SL2SSLIOP *mode*	orbasec.ssliop=*mode*
-SL2SSLIOPHost *host*	orbasec.ssliop.host=*host*
-SL2SSLIOPPort *port*	orbasec.ssliop.port=*port*
-SL2IIOP *mode*	orbasec.iiop=*mode*
-SL2IIOPHost *host*	orbasec.iiop.host=*host*
-SL2IIOPPort *port*	orbasec.iiop.port=*port*
-SL2KerberosSession	orbasec.kerberos_session=true
-SL2AnonymousSSL	orbasec.anonymous_ssl=true
-SL2AllowIIOP	orbasec.allow_iiop=true

TABLE 3. ORBASEC command-line options

corresponding ORBAsec property.

Secure ORB Services

You may specify ORB services using the ORBACUS **ooc.service** properties or the ORBACUS -*ORBservice* command-line option. However, references to *secure* ORB services must be established *after* credential acquisition via the **PrincipalAuthenticator**. Unfortunately, The **PrincipalAuthenticator** is only accessible after SL2 initialization, so ORBASEC SL2 requires a two-phase initialization in order to create secure references to ORB services. For this purpose the ORBASEC SL2 class provides a static method **add_initial_services**, which creates secure references to designated ORB services.

```
import org.omg.CORBA.*;
import java.util.Properties;
import orbasec.SL2;
public void main(String[] args)
{
  SL2.init_with_boa( args, new java.util.Properties() );
  ORB orb = SL2.orb();
  BOA boa = SL2.boa();

  // authenticate
  ...

  // create references to secure ORB services
  SL2.add_initial_services();
  ...
}
```

Getting SecurityCurrent

During the initialization process a service is created called **SecurityCurrent**, and it installed on the ORB. You get the reference to the **SecurityCurrent** object by using the `resolve_inital_references`call, which is illustrated by the following code fragment:

```
// Java
public void main(String[] args)
{
  ...
  // ORB and possibly BOA initialization
```

```
  // SL2 initialization

  org.omg.CORBA.Object obj =
    orb.resolve_initial_references("SecurityCurrent");

  org.omg.SecurityLevel2Current current =
    org.omg.SecurityLevel2.CurrentHelper.narrow(obj);

  ...
}
```

Note – A reference to the Security Current object cannot be obtained before ORBASEC SL2 is initialized.

Adding your own Security Mechanisms

The **CORBA::SecurityReplaceable** module was designed with the intention of allowing vendors to replace security components suitable for distribution in accordance with export restrictions specific to a country or locality. Beginning with ORBASEC SL2 Beta 3, Application Programmers can provide their own **SecurityReplaceable** security components and use them with ORBASEC SL2, allowing pluggable security mechanism components within ORBASEC SL2.

Assuming you have written your own implementation of the **SecurityReplaceable** module, you can use your implementation of these interfaces with ORBASEC SL2 by providing an implementation of the **orbasec.corba.SECIOPMechanismInitializer** interface, defined as follows:

```
package orbasec.corba;
public interface SECIOPMechanismInitializer
{
    public void
    init(
        org.omg.CORBA.ORB      orb,
        org.omg.CORBA.BOA      boa,
        java.util.Properties   properties );

    public org.omg.SecurityReplaceable.Vault
    get_vault();
}
```

This interface defines the following methods:

init

This method will be called during ORBASEC SL2 initialization with the **ORB** and **BOA** that were created in one of the **orbasec.SL2** initializers. The **properties** parameter will be derived from properties established during initialization. See See "ORBAsec SL2 Configuration" on page 47. for a description about the precedence tules governing the definition of these properties.

get_vault

This method must return the **org.omg.SecurityReplaceable.Vault**, from which the rest of the **SecurityReplaceable** relevant components (Credentials, SecurityContext, etc.) are obtained. You should return a reference to the **Vault** you have implemented.

To notify ORBASEC SL2 of the **SECIOPMechanismInitializer** you have defined, you must then specify the fully qualified class name of the initializer in an ORBASEC SL2 property of the form:

orbasec.seciop.mechanism_initializer.<*mechanism_name*>

where <*mechanism_name*> is a name you may choose to distinguish different **SECIOPMechanismInitializer**s you might install. The *value* of this property should be the fully qualified class name of the **SECIOPMechanismInitializer** you have defined.

Note – You may choose any mechanism name for this property, as long as it does not conflict with any other mechanism name you have defined for the same session. There are no ORBASEC "reserved" names, and any name you choose has no significance to ORBASEC SL2.

For example, if you have written a **SECIOPMechanismInitializer** called `com.acme.MechanismInitializer`, then you would write the following property into the configuration file:

```
orbasec.seciop.mechanism_initializer.my_initializer=\
    com.acme.MechanismInitializer
```

During ORBASEC SL2 initialization, the specified **SECIOPMechanismInitializer** will be loaded and an instance of it will be created with its default constructor. Then the **init** and **get_vault** methods of this class will be called. The **Vault** will be registered with ORBASEC SL2, and subsequent calls to the **PrincipalAuthenticator**'s **authenticate** method will acquire credentials using the specified **Vault**.

Note – Calls to **authenticate** should use the fully qualified mechanism name (i.e., with the provider) in the **mechanism** parameter in order for ORBASEC SL2 to select your **Vault**.

You should define a **orbasec.seciop.**<*mechanism_name*>**. mechanism_initializer** property for each <*mechanism_name*> defined in the **orbasec.seciop.mechanisms** property. This way, you can use any number of security mechanisms with ORBASEC SL2.

There is no need to specify security mechanisms and **SECIOPMechanismInitializer**s for the default SL2-GSSKRB SECIOP security mechanism. ORBASEC SL2 will attempt to load this module by default during initialization.

SL2 Version

The **orbasec.SL2** class provides a static String attribute called **Version**, which can be used to obtain a String representation of the current version of ORBASEC SL2.

```
public static final String Version;
```

You can print this String to the screen by running an ORBASEC SL2 enabled application with the **-SL2Version** flag at the command line. With this flag set, the application will print the version to the screen and exit.

```
prompt% java <my_app_name> -SL2Version
ORBAsec SL2 2.0.0
```

Equivalently, you may simply run the **main** method of the **orbasec.SL2** class.

```
prompt% java orbasec.SL2
ORBAsec SL2 2.0.0
```

CHAPTER 4

Security Current

Security Current

The Security Level 2 **Current** object is a locality constrained CORBA object that maintains state information associated with the current execution context, such as in a multi-threaded execution model. This information is specific to the current thread of execution and the process/capsule to which the thread belongs.

Getting the Current Object

The **SecurityLevel2::Current** object is returned from a call to the ORB's **resolve_initial_references** operation using the name "SecurityCurrent".

```
// Java
org.omg.CORBA.ORB orb = // The SL2 initialized ORB;
org.omg.SecurityLevel2.Current current =
   org.omg.SecurityLevel2.CurrentHelper.narrow(
           orb.resolve_initial_references("SecurityCurrent")
   );
```

ORBASEC SL2 Extentions to Current

ORBASEC SL2 makes several extensions to **SecurityLevel2::Current**. The operations and attributes that are beyond the standard **SecurityLevel2::Current** interface are defined in an ORBASEC SL2 definition of the **SecLev2::Current** interface. This interface is explained at the end of this chapter under "ORBAsec SL2 Extensions to Current" on page 67.

Standard Attributes and Operations

The attributes and options on the **SecurityLevel2::Current** object are described below along with their values, semantics, and possible restrictions as pertaining to the ORBASEC SL2 implementation.

The following operations and attributes are standard on **SecurityLevel2::Current**.

supported_mechanisms

This attribute returns a list of **Security::MechandOptions** structures. Each element in the list gives the mechanism available and the **Security::AssociationOptions** the mechanism supports.

```
//IDL
readonly attribute Security::MechandOptionsList
                                        supported_mechanisms;

// Java
public org.omg.Security.MechandOptions[]
                                      supported_mechanisms();
```

This attribute may be examined to select the security mechanism available.

```
// IDL
module Security {
struct MechandOptions {
    MechanismType          mechanism_type;
    AssociationOptions     options_supported;
};
};
```

```
// Java
package org.omg.Security;
final public class MechandOptions {
    String                  mechanism_type;
    short                   options_supported;
}
```

In ORBASEC SL2, Mechanism strings have a particular structure. The structure is:

<mechanism_type> :: <mechanism_identifier> [‘,’ <ciphersuite>]
<mechanism_identifier> :: <mechanism>’_’<provider>

The first component of the mechanism type identifier is the name of the security mechanism. The further components, separated by commas, are the cipher suites. All cipher suites have symbolic names.

Examples of some mechanism strings supported by ORBASEC SL2 are:

```
“Kerberos”
“Kerberos_MIT”
“SSL,DH_DSS_3DES_CBC_MD5,DHE_DSS_DES_CBC_SHA”
“SSL_IAIK,DH_anon_DES_CBC_MD5”
```

The string “Kerberos” can be used to authenticate a Kerberos principal, which creates a credentials object using the Kerberos infrastructure using an implementation from the default provider. The string “Kerberos_MIT” can be used to further stipulate that a certain provider be used, namely ORBASEC SL2-GSSKRB “plug-in”. (MIT means the Kerberos implementation from M.I.T.) The string “SSL,DH_DSS_3DES_CBC_MD5,DHE_DSS_DES_CBC_SHA” can be used to authenticate principal using a Public Key Infrastructure (PKI), which creates a credentials object with the ability to use SSL with the listed cipher suites. The string starting with “SSL_IAIK” means to use the ORBASEC SL2-SSL “plug-in”, which uses the SSL toolkit from IAIK.

Note – It may not be possible to have two different providers for one mechanism in the same ORB, although we have not yet experimented with this capability.

received_credentials

This read-only attribute is valid only in the context of an object servicing a request on the target side. Its value is thread specific. It is meant to represent the security context that has been established between the target’s own credentials and the cli-

ent's own credentials. Therefore, it represents the identity of the client and any other security attributes the client may have delivered to the target.

```
// IDL
readonly attribute ReceivedCredentials   received_credentials;

// Java
public org.omg.SecurityLevel2.ReceivedCredentials
                                          received_credentials();
```

Accessing this attribute while not in the context of servicing an object request, such as in a pure client application will result in the raising of a **CORBA::BAD_OPERATION** exception.

own_credentials

This attribute is the list of credentials that have been created and initialized by the application using the **PrincipalAuthenticator** object. Its value is capsule specific, meaning the "own" credentials are owned by the capsule that authenticated them. A facility called **remove_own_credentials** can remove certain credentials from the list.

```
// IDL
readonly attribute CredentialsList       own_credentials;

// Java
public org.omg.SecurityLevel2.Credentials[]
                                         own_credentials();
```

The capsule may own or initialize any number of credentials using the **Principal-Authenticator** object. In fact, the only way a **Credentials** object makes it on the "own" credentials list, is by way of the **PrincipalAuthenticator** object. This object is described in "Principal Authenticator" on page 75.

Once a **Credentials** object is created by the **PrincipalAuthenticator** object, it is placed on the "own" credentials list only after the **Credentials** object becomes fully initialized (depending on the mechanism and authentication method, principal authentication may be a multistep process). The **Credentials** object remains on the "own" credentials list until it is removed by application using the **remove_own_credentials** operation. It is the responsibility of the application to remove **Credentials** objects from the "own" credentials list when they expire, or when they have become invalid. Removal from the "own" credentials list does not happen automatically.

remove_own_credentials

This operation removes a given **Credentials** object from the own credentials list.

```
// IDL
void remove_own_credntials(
    in     Credentials        creds;
);

// Java
public void remove_own_credentials(
    org.omg.SecurityLevel2.Credentials   creds
);
```

This operation gives the programmer some management over the “own” credentials list, should the application authenticate many principals. The application is responsible for removing **Credentials** objects from the “own” credentials list when they have become invalid or expired. Removal does not happen automatically.

principal_authenticator

This attribute’s value is the **PrincipalAuthenticator** object that is available in the environment. This attribute is capsule specific and is a read-only attribute. It is used by the application to authenticate principals, which create “own” type Credentials objects that represent that principal. This object is described in [“Principal Authenticator” on page 75].

```
// IDL
readonly attribute PrincipalAuthenticator
                                     principal_authenticator;

// Java
public org.omg.SecurityLevel2.PrincipalAuthenticator
                                    principal_authenticator();
```

get_security_mechanisms

This operation for clients wishing to determine which security mechanisms that are associated with a target object reference. It returns a list of all the security mechanisms, which are structures containing security names and required/supported option pairs that are contained in the object’s IOR.

```
//IDL
Security::SecurityMechanismDataList get_security_mechanisms(
    in   Object obj_ref
);

//Java
public org.omg.Security.SecurityMechanismData[]
get_security_mechanisms(org.omg.CORBA.Object obj_ref);
```

This operation returns a structure containing the security mechanism, security name, association options that the target object requires and supports. The list contains security mechanism and names contained in the objects IOR. These names may be different than the authenticated name, which is in the AccessId attribute of the received credentials. Duplicates are not removed from this list.

```
// IDL
module Security {
struct SecurityMechanismData {
    MechanismType                   mechanism;
    Opaque                          security_name;
    AssocationOptions               options_supported;
    AssociationOptions              options_required;
};
};

//Java
package org.omg.Security;
final public class SecurityMechanismData {
    String                           mechanism;
    byte[]                           security_name;
    short                            options_supported;
    short                            options_required;
}
```

get_target_credentials

This operation is for clients wishing to check the authentication of the target object reference. It returns a received credentials object containing the attributes of the object..

```
//IDL
TargetCredentials get_target_credentials(
    in   Object target
);
```

```
//Java
public org.omg.SecurityLevel2.TargetCredentials
get_target_credentials(org.omg.CORBA.Object target);
```

This operation returns a **TargetCredentials** object with attributes of the security context between the client and the server of this object.

get_policy

This operation is meant to return the policies placed on the **Current** object that are the default policies for making invocations on objects that do not have their own policy overrides set.

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

// Java
public org.omg.CORBA.Policy get_policy(int policy_type);
```

Unfortunately, this call is semantically defined in the CORBA CORE to retrieve the policy overrides on the object itself (i.e. the **Current** object), not as a repository for policies to be applied elsewhere. Therefore, we feel that this call is semantically wrong. However, we do support the **get_policy** call. However, it makes a call to **get_overrides** which will be the proposed **CORBA::PolicyCurrent** operation for this purpose. We have placed the **get_overrides** operation on the **orbasec::SecLev2::Current** object, which are ORBASEC SL2 extensions to the **SecurityLevel2::Current** object. See “ORBAsec SL2 Extensions to Current” on page 67.

required_rights_object

This attribute’s value is the **RequiredRights** object available in the environment. This attribute is capsule specific and is a read-only attribute. This object is stated to be used rarely by any application; it is generally used by any **AccessDecision** objects to find the rights required to use a particular interface. However, it may be used by applications if the application wants to implement its own access control.

```
// IDL
readonly attribute RequiredRights
                                    required_rights_object;
```

```
// Java
public org.omg.SecurityLevel2.RequiredRights
                                        required_rights_object();
```

Since this version of ORBASEC SL2 does not support automatic access control, accessing this attribute raises a **CORBA::NO_IMPLEMENT** exception.

access_decision

This attribute's value is the **AccessDecision** object available in the environment. This attribute is capsule specific and is a read-only attribute. It is used to make access decisions on invocations on interfaces. It may have any implementation, but is supposed to interact with **Credentials** objects, **RequiredRights** objects, and **DomainAccessPolicy** objects.

Note – It is not well defined in the security specification on the topic of the number of **RequiredRights** objects that can exist in a capsule and the number of **AccessDecision** objects that can exist in a capsule. However, it would imply by this attribute that only one access decision object may exist.

```
// IDL
readonly attribute AccessDecision
                                        access_decision;

// Java
public org.omg.SecurityLevel2.AccessDecision
                                        access_decision();
```

Since this version of ORBASEC SL2 does not support automatic access control, accessing this attribute raises a **CORBA::NO_IMPLEMENT** exception.

audit_decision

The attribute's value is the **AuditDecision** object available in the environment. This attribute is capsule specific and is a read-only attribute. It is suppose to be used by the application to obtain information about what needs to be audited for other specific object/interface in this environment.

Note – Again, it is not well defined on the topic of the number of AuditDecision objects should exist and for what purpose, and which AuditChannel objects should exist.

```
// IDL
readonly attribute AuditDecision     audit_decision;

// Java
public org.omg.SecurityLevel2.AuditDecision
                                     audit_decision();
```

Since this version of ORBASEC SL2 does not support automatic access control, accessing this attribute raises a **CORBA::NO_IMPLEMENT** exception.

ORBASEC SL2 Extensions to Current

ORBASEC SL2 makes several extensions to the **SecurityLevel2::Current** object. The extensions to **SecurityLevel2::Current** come in the form of a and interface called **SecLev2::Current** that inherits from **SecurityLevel2::Current**.

The IDL definition of the ORBASEC SL2 **SecLev2::Current** interface is below:

```
//IDL
#include <SecurityLevel2.idl>
#pragma prefix “orbasec”

module SecLev2 {
  interface Current : SecurityLevel2::Current {
    // Policy Operations
    void set_overrides(
       in CORBA::PolicyList           policies,
       in CORBA::SetOverrideType      override_type
    );

    CORBA::PolicyList get_overrides(
       in CORBA::PolicyTypeSeq        policy_types
    );

    void remove_overrides(
      in CORBA:PolicyTypeSeq          policy_types
    );

    // Accepting Credentials Operations
    attribute SecuirtyLevel2::CredentialsList
                                      accepting_credentials;

    void set_accepting_credentials(
      in Object                            servant,
      in SecurityLevel2::CredentialsList   creds_list
    );

    SecurityLevel2::CredentialsList get_accepting_credentials(
       in Object                           servant
    );

    void release_accepting_credentials(
      in Object                            servant
    );
  };
};
```

The **SecLev2::Current** object is returned from a call to the ORB’s **resolve_initial_references** operation using the name “SecurityCurrent”.

```
// Java
org.omg.CORBA.ORB orb = // The SL2 initialized ORB;
orbasec.SecLev2.Current current =
   orbasec.SecLev2.CurrentHelper.narrow(
           orb.resolve_initial_references("SecurityCurrent")
   );
```

The **SecLev2::Current** object contains two sets of operations. The first set pertain to setting and getting security related policies with respect to the current thread of execution. The second set pertains to assigning the proper credentials objects with servant object references, so that the proper credentials information gets placed in the object's IOR when it is exported to potential clients.

Policy Operations

The following operations are the proposed (or about to be proposed) operations of the **CORBA::PolicyManager** interface, which the **CORBA::PolicyCurrent** interface inherits. It may be proposed that the **SecurityLevel2::Current** inherit the **CORBA::PolicyCurrent** interface for handling of thread based policies. Both the **CORBA::PolicyManager** and **CORBA::PolicyCurrent** interfaces are yet to be adopted and are part of the CORBA Messaging RFP response.

Policies set with these interfaces apply as default policies to object references that are introduced into the current thread of execution. Object references are introduced into a current thread of execution by unmarshalling an IOR. This unmarshalling is done by the **ORB::object_to_string** operation or automatically by getting an object reference from an invocation, such as getting an object reference from a naming service.

set_overrides

This operation is thread specific and adds or sets the given policies for the current thread of execution.

```
// IDL
void set_overrides(
       in CORBA::PolicyList           policies,
       in CORBA::SetOverrideType      override_type
);

// Java
public void set_overrides(
    org.omg.CORBA.Policy[]    policies,
    int                       override_type
);
```

Policies that are set on the thread can be overridden further and more specifically on an object by using the **set_policy_overrides** pseudo operation on an object reference.

get_overrides

This operation is thread specific and gets the policies named by the given policy types for the current thread of execution.

```
// IDL
CORBA::PolicyList get_overrides(
       in CORBA::PolicyTypeSeq   policy_types
);

//Java
public org.omg.CORBA.Policy[] get_overrides(
    int[]                        policy_types
);
```

The policy objects returned from this call applies first to the current thread of execution. If a policy of a given type has not been set on the current thread of execution specifically, a policy set at the original initializing thread of execution is retrieved. This allows a server application to set policies for the server at initialization time before readying the object to start accepting requests. This procedure is most useful in the thread-per-request execution model.

If a given policy type is not available, (i.e. set at the thread or the initializing thread) no policy of that type is returned in the list. No exception is raised.

remove_overrides

This operation removes policies of the given types that have been set on the current thread of execution.

```
// IDL
void remove_overrides(
      in CORBA:PolicyTypeSeq policy_types
);

// Java
public void remove_overrides(
     int[]                  policy_types
);
```

If policies of a given type are not found to be previously set on the current thread of execution, the operation ignores it, and continues with the removal process for any other policy types given to the operation. No exception is raised.

Accepting Credentials Attributes and Operations

The following section contains descriptions for associating a set of credentials objects with a servant object. A servant object is an implementation of a **CORBA::Object** that is local to the capsule. Associating Credentials with a servant object tells the security service which security mechanism components, which are derived from the **Credentials** objects, to place in the servant object's IOR.

accepting_credentials

This thread specific attribute sets or returns the list of Credentials objects that are associated with new object references, (i.e. object implementations that are connected to the ORB) within the current thread of execution. The credentials in the **accepting_credentials** attribute are set to be the default for the current thread of execution. To override those defaults for a particular servant object the call **set_accepting_credentials** must be called on that servant object implementation.

Note – Calling **set_accepting_credentials** on an object connects the object to the ORB. The object must not be previously connected to the ORB, as that is the point at when its IOR gets generated. If an object given to **set_accepting_credentials** is already connected to the ORB a **CORBA::BAD_PARAM** exception is raised.

The initial value for **accepting_credentials** is the entire list of **own_credentials** at the time the servant is connected to the BOA.

```
// IDL
attribute SecurityLevel2::CredentialsList
                                        accepting_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
             accepting_credentials();

public void   accepting_credentials(
     org.omg.SecurityLevel2.Credentials[] creds
);
```

If an attempt to set this attribute to a list of credentials containing a **Credentials** object without the ability to accept secure associations, such as a Credentials object with no **accepting_options_supported**, then a **CORBA::BAD_PARAM** exception is raised. An example of such a **Credentials** object might be the **ReceivedCredentials** object from the **received_credentials** attribute, or a **TargetCredentials** object retrieved from the **get_target_credentials** operation.

set_accepting_credentials

This operation sets the credentials to be used as the authenticating credentials for a particular servant object.

The given object should be not yet be connected to the ORB! If the object is previously connected to the ORB the accepting credentials that were associated with the thread at the time of the connect are the accepting credentials that are associated with the object reference. Therefore, this operation would have no effect, and hence a **CORBA::BAD_PARAM** exception will be raised. **This operation will connect the object to the ORB.**

```
// IDL
void set_accepting_credentials(
      in Object                              servant,
      in SecurityLevel2::CredentialsList   creds_list
);

// Java
public void set_accepting_credentials(
   org.omg.CORBA.Object                    servant,
   org.omg.SecurityLevel2.Credentials[]   creds_list
);
```

If an attempt to use this operation with a list of credentials containing a **Credentials** object without the ability to accept secure associations then a **CORBA::BAD_PARAM** exception is raised. If an attempt to use this operation on an object that is not a servant, a **CORBA::BAD_PARAM** exception is raised.

get_accepting_credentials

This operation is used to retrieve the accepting credentials that have been set at the authenticating credentials for a particular servant object when the servant object was connected to the ORB. This object must be connected to the ORB, or else a **CORBA::BAD_PARAM** exception will be raised.

```
// IDL
SecurityLevel2::CredentialsList get_accepting_credentials(
       in Object               servant
);

// Java
public org.omg.SecurityLevel2.Credentials[]
get_accepting_credentials(
    org.omg.CORBA.Object       servant
);
```

If no credentials were set for the given servant object, an empty sequence is returned. If an attempt to use this operation on an object that is not a servant, a **CORBA::BAD_PARAM** exception is raised.

release_accepting_credentials

This operation is used to remove the association of accepting credentials with the given servant object. This object must be connected to the ORB, or else a **CORBA::BAD_PARAM** exception will be raised.

```
// IDL
void release_accepting_credentials(
      in Object              servant
);

// Java
public void release_accepting_credentials(
    org.omg.CORBA.Object     servant
);
```

CHAPTER 5

Principal Authenticator

Principal Authenticator

This section describes the application programmer's use of the **SecurityLevel2::PrincipalAuthenticator** interface and its specific implementation relating to the default **SecurityReplaceable** module installed in ORBASEC SL2.

The PrincipalAuthenticator interface is implemented by a sole principal authenticator object. This object is a capsule specific object that resides on the Security Current object. It is retrieved as follows:

```
// Java
org.omg.SecurityLevel2.Current current = // ... get current
org.omg.SecurityLevel2.PrincipalAuthenticator pa =
                              current.principal_authenticator();
```

For details on the mechanism to get the Security Current object, see "Getting the Current Object" on page 59.

An application programmer uses the **PrincipalAuthenticator** object to initialize the application's "own" credentials. The principal authenticator makes calls on the vault behind the application programmer's view. It asks the vault to authenticate a specific principal and create the **Credentials** object that represents that principals

identity. We term this notion as the “acquisition” of credentials. The **PrincipalAuthenticator** object then places these operational credentials on the **Current** object for retrieval by the application programmer.

The **PrincipalAuthenticator** interface has three basic operations, **get_supported_authen_methods**, **authenticate**, and **continue_authentication**. The **get_supported_authen_methods** operation takes a security mechanism identifier and returns the list of authentication methods that are available for that mechanism. The **authenticate** operation starts an authentication sequence that may take several steps. If additional steps are needed to complete authentication of the principal, the **continue_authentication** operation is used for as many times as needed.

A **Credentials** object caught in a multistep authentication process contains state information to facilitate the continuation of the authentication process. Once successfully completed, indicated by a **Security::AuthenticationStatus** enum value of **SecAuthSuccess** returned by the **PrincipalAuthenticator** object, the created **Credentials** object is placed on the **SecurityLevel2::Current** object’s “own” credentials list.

The operations for the **PrincipalAuthenticator** object’s interface are defined below:

authenticate

This operation begins the authentication of a principal. We say begin, because authentication may take several steps to complete, such as with a challenge/response oriented mechanism. The **authenticate** operation’s interface is described below.

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

```
// Java
public org.omg.Security.AssocationStatus
authenticate(
    int                                     method,
    String                                  mechanism,
    byte[]                                  security_name,
    byte[]                                  auth_data,
    org.omg.Security.SecAttribute[]         privileges,
    org.omg.SecurityLevel2.CredentialsHolder
                                            creds,
    org.omg.Security.OpaqueHolder           continuation_data,
    org.omg.Security.OpaqueHolder           auth_specific_data
);
```

The parameters to the **authenticate** operation are described below:

method

This parameter specifies the authentication method that will be used to authenticate the principal. Values for the authentication method are parameterized on the mechanism selected. These values are returned from a call to the **get_supported_authen_methods** operation, which takes the mechanism as an argument.

No values for this parameter have been specified by the OMG presently. Therefore, we only accept the value 0, meaning “the default for the mechanism”.

mechanism

This parameter specifies the mechanism with which to authenticate the principal with and create its associated “own” type credentials. The mechanisms that are allowed in this call are the mechanisms that are listed as supported mechanisms from the call to **SecurityLevel2::Current** object’s **get_supported_mechanisms** attribute.

security_name

This parameter is a byte array stating the recognized name of the principal to be authenticated. The contents and its encoding into bytes of this parameter is specific to the mechanism specified. For some mechanisms, this parameter may be an empty sequence of bytes. The name supplied here must be a

orbasec.corba.Opaque byte encoded name. Please see Chapter on "Opaque Encodings" on page 185 for details.

If you have the ORBASEC SL2-GSSKRB distribution and you are using a Kerberos mechanism, a kerberos security name, such as "bart@MYREALM.COM" must be represented as the following:

```
// For Kerberos
byte[] security_name =
            orbasec.corba.Opaque.encodeKerberosName(
                          "bart@MYREALM.COM").getEncoding();
```

If you have the ORBASEC SL2-SSL distribution and you are using an SSL mechanism, the security name must be an empty sequence since the default authentication mechanism gets the security name from the certificate file named in the **auth_data** parameter.

```
// For SSL
byte[] security_name = new byte[0];
```

auth_data

This parameter specifies the extra data needed to authenticate the principal. The format of this object is a sequence of bytes and the format of the data is dependent on the mechanism and the method used. A value of an argument to this parameter may contain such esoteric data as the result of a fingerprint or retinal scan.

If you have either of the ORBASEC SL2-GSSKRB or ORBASEC SL2-SSL distributions the value of the **auth_data** parameter be a byte encoded Java string of the form read by the **java.util.Properties** class. This form mandates a "name=value" string format separated by newline characters. The name-value pairs that are required and their meaning are listed at the end of the section. The names of the attributes and their associated values are explained at the end of this chapter for both SL2-GSSKRB and SL2-SSL

privileges

This parameter states the "extra" privileges that the application programmer wants to be authenticated along with the principal to create the credentials with those privileges authorized. Such privileges can have values stating facts such that whether the principal is the member of a group or has the authorization for a particular role.

Note – Currently, in both the ORBAsec SL2-GSSKRB and ORBAsec SL2-SSL distributions, this field is ignored, as neither mechanism can handle the authentication or authorization of privileges in this manner. However, future mechanisms may have this capability.

creds

This parameter is an output parameter returning the newly created **Credentials** object of the "own" type. This operation works in concert with the **Current** object and places the new credentials in the current's own credentials list should the return value from authenticate be **SecAuthSuccess**. If it is **SecAuthContinue** the **Credentials** object may not be fully enabled. The authentication mechanism created interim credentials to be further passed to the **continue_authentication** operation.

In both the ORBAsec SL2-GSSKRB and ORBAsec SL2-SSL distributions authentication is a one step process. The authenticate call either returns **SecAuthSuccess** and places the fully enabled **Credentials** object on the Current object's "own" credentials list, or it raises a system exception with an informative message.

continuation_data

This parameter is an output parameter returning data needed to continue the authentication. This may hold such data labeling a continuation context.

In both the ORBAsec SL2-GSSKRB and ORBAsec SL2-SSL distributions authentication is a one step process. The value of the **continuation_data** parameter is unaffected.

auth_specific_data

This parameter is an output parameter returning data that may need to be exposed to the application programmer, such as a message about what is needed to continue the authentication. It is completely mechanism and method specific.

In both the ORBAsec SL2-GSSKRB and ORBAsec SL2-SSL distributions authentication is a one step process. The value of the **auth_specific_data** parameter is unaffected.

return value

The value returned from this operation is one of the **Security::AuthenticationStatus** enumeration type and states whether authentication succeeded, failed, or needs to be continued.

In both the ORBASEC SL2-GSSKRB and ORBASEC SL2-SSL distributions authentication is a one step process. The authenticate call either returns **SecAuthSuccess** and places the fully enabled **Credentials** object on the Current object's "own" credentials list, or it raises a system exception with an informative message.

continue_authentication

This operation is meant to continue authentication steps started by **authenticate** or from previous calls to **continue_authentication**. Its interface is defined below:

```
Security::AuthenticationStatus continue_athentication(
    in   Security::Opaque      response_data,
    in   Credentials           creds,
    out  Security::Opaque      continuation_data,
    out  Security::Opaque      auth_specfic_data
);
```

In both the ORBASEC SL2-GSSKRB and SL2-SSL distributions authentication is a one step process. The **authenticate** call does not return **SecAuthContinue**. Calls to this operation raises a **CORBA::BAD_OPERATION** exception.

response_data

This parameter returns data in the format required by the mechanism and method for continuing authentication. Its authenticate counterpart is the **auth_data** parameter.

creds

This parameter should be credentials returned from **authenticate** or subsequent calls to **continue_authentication**. If the operation returns **SecAuthSuccess**, the credentials will be fully enabled and placed on **Current**'s own credentials list.

continuation_data

This parameter should be continuation data returned from **authenticate** or subsequent calls to **continue_authentication**. If the operation returns **SecAuthContinue**, this output value should be used in the subsequent call to **continue_authentication**.

auth_specific_data

This parameter should be authentication specific data returned from **authenticate** or subsequent calls to **continue_authentication**. If the operation returns **SecAuthContinue**, this output value should be used in the subsequent call to **continue_authentication**.

get_supported_authen_methods

This operation returns a sequence of authentication methods that are valid for the calls to authenticate. At some point there will be a standard set. The authentication methods are parameterized on the mechanism, as some methods may only be valid authentication methods for particular mechanisms. Its interface is defined below:

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

Note – Currently, only one authentication method is supported for both the ORBASEC SL2-GSSKRB and ORBASEC SL2-SSL distributions, indicated by the integer 0.

Authentication using ORBASEC SL2-GSSKRB

This section explains the mechanisms, security name, and authentication data formats for using the PrincipalAuthenticator with the ORBASEC SL2-GSSKRB distribution. This distribution gives you the ability to use standard GSS-API version of Kerberos as defined by MIT.

Mechanism

If you have the ORBASEC SL2-GSSKRB distribution, you can currently only specify one mechanism for authenticate. It is:

```
Kerberos_MIT
```

or its default companion:

```
Kerberos
```

In ORBASEC SL2 mechanism naming scheme, the latter two match the above one. For now, our SL2-GSSKRB distribution only has support for the cipher suites with your Kerberos installation. There is currently no way to specify them.

Security Name

The **security_name** parameter must be in a **orbasec.corba.Opaque** byte encoding of a Kerberos name. This requires taking the string of a kerberos principal such as "bart@MYREALM.COM" and converting it to a special **Opaque** byte encoding. This encoding is simply done as:

```
import orbasec.corba.Opaque;

byte[] security_name =
 Opaque.encodeKerberosName("bart@MYREALM.COM").getEncoding();
```

If the **security_name** parameter has one of the following values:

- `new byte[0]`
- `Opaque.encodeKerberosName("").getEncoding()`

The name stored in the named or default Kerberos credentials cache will be used.

Authentication Data

The authentication data is the value of the **auth_data** parameter for the **authenticate** operation. The format for this parameter is the standard Java string to byte encoding of a Java String containing name-value pairs in the form for the **java.util.Properties** class. This format requires each entry to have the form "name=value" separated by newline characters.

For example, a call in Java to authenticate "bart@MYREAL.COM" would be:

```
import org.omg.Security.*;
import org.omg.SecurityLevel2.*;
import orbasec.corba.Opaque;

....
    PrincipalAuthenticator pa = // get the PA from Current

    CredentialsHolder      credsh = new CredentialsHolder();
    OpaqueHolder           contdata = new OpaqueHolder();
    OpaqueHolder           authspecdata = new OpaqueHodler();
    AuthenticationStatus   stat;

    stat = pa.authenticate(
                0,
                "Kerberos",
                Opaque.encodeKerberosName("bart@MYREAL.COM")
                          .getEncoding(),
                ("config=FILE:/etc/krb5.conf\n"   +
                 "password=\"MyPassword\"\n" +
                 "cache=FILE:/tmp/krb5cc_bart\n"   +
                 "lifetime=360m\n"           +
                 "forwardable=true\n").getBytes(),
                new SecAttribute[0],
                credsh,
                contdata,
                authspecdata
    );
```

The names of the properties that are valid for the authentication data are described below.

config

This property field contains the name of the Kerberos configuration file. The existence of this file is specific to the MIT implementation. This file contains information pertaining to the configuration of the kerberos configuration. Such information includes the network location of the KDC, and other parameters. If the **config** field is not present or its value is empty, the default of the Kerberos installation is used.

The config specification has a two part format:

```
<type>:<config name>
```

However, the only type that is currently valid, is "`FILE`", where the config name part names the location of a Kerberos Version 5 configuration file on the local system.

On most Unix systems, the default configuration file for Unix systems is located by the name `/etc/krb5.conf`, or by the contents of an environment variable called "`KRB5_CONFIG`".

On NT, the default configuration file is specified by a complex logic.

The "kerberos.ini" file must be first located wherever "ini" files are found. This procedure may be some uniform directory search according to your system, such as "C:\winnt;C:\windows;\C:\winnt\system", etc.

This "kerberos.ini" file may contain an entry as follows:

```
[Files]
    krb5.ini = .....
```

If there is no "krb5.ini" entry, it assumes that "krb5.ini" file exists in your current directory.

password

This property field must contain the password. Unfortunately, we have a very minor character translator, so special characters like control characters, tabs, backspaces, and such are not representable. If the **password** field is not present or its value is empty, and the **keytab** field (see below) is not present or is empty, then the Kerberos authentication mechanism will use the credentials in the specified **credentials cache** file (see below). This assumes the principal has already acquired credentials externally (for instance, via the Kerberos *kinit* program). If the **keytab** field is present and non-empty, then the principal's key is assumed to be stored in the **keytab** file.

cache_name

This field names the location of Kerberos credentials **cache** that you want to use. If the **password** and **keytab** fields are not present, the **cache** file should contain the principal's already-obtained credentials (e.g., via the Kerberos *kinit* program). The credentials cache specification has a two part format:

```
<type>:<credentials ccache name>
```

The type can be one of "FILE" or "MEMORY". The "FILE" type names a file on the local system, meaning that the credentials are or will be placed externally to the running process. If the type is "MEMORY" the credentials will be retained inside the process for the duration of the process. Using a credentials cache of type "MEMORY" is a safer way to go. However, to use the "MEMORY" type credentials cache, you must supply a value for the **password** or **keytab** field, as it is impossible to use already authenticated credentials.

Also, if authenticating multiple Kerberos Credentials and using "MEMORY" type credential caches, the names must be different. When using "MEMORY" type caches the name portion of the cache file is not that significant, so names such as MEMORY:0, MEMORY:1, MEMORY:2, and so on, can be used without any difficulty.

If the cache file specification is not present, then the default "kerberos session" cache is assumed. The Kerberos session cache is a file, and it is usually initialized by the "kinit" program from the Kerberos distribution.

The default credentials cache file on most Unix systems, is "FILE:/tmp/krb5cc_<*uid*>" where *uid* is the user number of the principal logged on, or named by an environment variable "KRB5CCNAME".

On NT, it resides in a file, which may be specified in the "kerberos.ini" file one of two ways.

```
[Files]
    RegKRB55CCNAME = ....
```

or

```
[Files]
   krb5cc = ....
```

If the first method is used, which takes precedence over the first, the value names a registry key that points to the file name. Such a registry key might be "[HKEY_CURRENT_USER\Software\Gradient\DCE\Default\KRB5CCNAME]", and its value will contain a string with the "FILE:" prefix.

If the second method is used, the "krb5cc" variable names the cache file directory, using the "FILE:" prefix.

If the **security_name** parameter is nonempty and the **cache** specifies a “FILE” type, or the default “session” cache. The **security_name** must match the principal stored in the credentials cache file. If the **security_name** does not match the principal stored in the credentials cache file, a **CORBA::BAD_PARAM** exception is raised. If the **security_name** parameter is empty, then the principal name in the credentials cache file is used for the **Credentials** object.

enable_server

This property field is important if the credentials you are authenticating will be used to accept secure assoications. That is to say the process holding on the credentials is to be a server. If this property has a value of “true”, it signifies that a keytab will be used.

keytab

This property field is important and is especially important if the “enable_server” property is set to “true”. If you want your application to be a CORBA server, i.e. to service any requests on its objects from remote clients, (i.e. you initialize the BOA), you must have the key stored in a readable keytab. If the keytab you need is a file, which is given to you by your Kerberos administrator, giving a value to this field forgoes the use of **password**, because effectively the keytab contains the password.

The keytab specification has a two part format:

```
<type>:<keytab name>
```

To use a keytab file, the principal must be contained in a keytab file that the process has permission to read. Note, that in most Unix installations of the MIT Kerberos implementation the default keytab file (`/etc/krb5.conf`) is usually only readable by the super user, as this file contains the keys for such services as TELNET, FTP, etc. These services initially have super user privilege until they authenticate the client and downgrade their privileges.

This situation may require you to have the Kerberos administrator make a **keytab** file available for your particular server’s principal. For MIT Kerberos installations only the `kadmin` or `kadmin.local` programs are allowed to create and add the keys of the principals to a keytab file.

If you label the keytab to be of a MEMORY type keytab, such as MEMORY:0, then you need a **password**. The system loads that memory keytab with the princi-

pal's key after deriving it from the password. This mechanism alleviates the need to expose a principal's key to the file system.

If the "keytab" property is not defined, the system default file is assumed. On Unix systems, the default is found first by the value of the "KRB5_KTNAME" environment variable, the file specified in the specified Kerberos Configuration file (e.g. /etc/krb5.conf), in the following manner:

```
[libdefaults]
    default_keytab_name = .....
```

Lastly, it is defined to be "/etc/krb5.keytab" if no entry is found.

On NT, the default keytab file is found the same way through the specified Kerberos Configuration File. However, if no entry is found it is assumed to be "krb5.keytab" in your current working directory.

Note – Using a MEMORY type keytab is the preferred mechanism for servers.

Note – It has been discovered that if your Kerberos Administrator adds your principal's name to a keytab file, at least in the MIT system, its key is randomized and the password is effectively changed to some unknown value.

lifetime

This property field specifies the lifetime of the credentials. Its value is of the formation of an integer immediately suffixed with one of "s", "m", "h", or "d". The suffixes each specify seconds, minutes, hours, or days respectively. Absence of the **lifetime** field defaults to the system default.

proxiable

This property field specifies that the credentials will be proxiable. This statement means that your credentials are able to be forwarded to the target on an invocation for the target to create authentication tickets in your behalf.
The field's value is of the form of **true** or **false**. Absence of the proxiable field defaults to **false**. Please see the Internet RFC 1510[1] for an explanation of the details.

forwardable

This property field specifies that the credentials will be forwardable. This means that your credentials are able to be forwarded to the target on an invocation for the target to create authentication tickets in your behalf. Its value is of the form of **true** or **false**. Absence of this field defaults to **false**. Please see the Internet RFC 1510[1] for an explanation of the details.

renewablelife

This property field has the same format as the **lifetime** field. It specifies the amount of time the credentials can be renewed. Absence of the **renewablelife** field defaults the renewable life to the system default.

Session Credentials Example

For use as a pure client, where an application uses the ORB, but does not initialize the BOA, or need to give up references to internal objects (callbacks), the default Kerberos credentials can be used. A default credentials file is set up when a user initializes his/her Kerberos credentials cache file by using the Kerberos “kinit” program. This program initializes your credentials cache file by asking for your Kerberos principal name and password. The credentials cache file is known as the user’s Kerberos “session” cache, and it is usually set up when the user is logged in, and destroyed when the user logs out.

In ORBASEC SL2, to create a **Credentials** object using the Kerberos “session” credentials cache, the following example illustrates the method by which that is done.

```
import org.omg.Security.*;
import org.omg.SecurityLevel2.*;

....
    PrincipalAuthenticator pa = // get the PA from Current

    CredentialsHolder      credsh = new CredentialsHolder();
    OpaqueHolder           contdata = new OpaqueHolder();
    OpaqueHolder           authspecdata = new OpaqueHodler();
    AuthenticationStatus   stat;

    stat = pa.authenticate(
                0,                        // auth method
                “Kerberos”,               // mechanism
                new byte[0],              // security_name
```

```
                new byte[0],              // auth_data
                new SecAttribute[0],      // privileges
                credsh,                   // out Credentials
                contdata,
                authspecdata
    );
```

Note – Session Credentials cannot be used in a CORBA server to accept CORBA requests with kerberos, because the principal's key cannot be retrieved.

Authentication using ORBASEC SL2-SSL

This section explains the mechanisms, security name, and authentication data formats for using the **PrincipalAuthenticator** with the ORBASEC SL2-SSL distribution. This distribution gives you the ability to use standard Secure Socket Layer Version 3.0. It uses the **iSaSiLk** toolkit from **IAIK**.

Mechanism

If you have the ORBASEC SL2-SSL distribution, you can specify one or more of many mechanisms (cipher suites) available for SSL. A mechanism is a string representation with the security mechanism name, plus cipher suites separated by commas. The IAIK toolkit has support for the following cipher suites:

SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_WITH_RC4_MD5
SSL_RSA_WITH_RC4_SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
SSL_RSA_WITH_IDEA_CBC_SHA
SSL_RSA_WITH_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_DSS_WITH_DES_CBC_SHA
SSL_DH_DSS_WITH_3DES_CBC_SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_RSA_WITH_DES_CBC_SHA
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA

SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
SSL_DH_anon_WITH_RC4_MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA

The above names are the symbolic names for the cipher suites. An example of mechanism name to use would be:

"SSL_IAIK,SSL_DH_DSA_WITH_DES_CBC_SHA"

The above mechanism name specifies the SSL mechanism from the IAIK provider (currently the only one). Similarly, the string:

"SSL,SSL_DH_DSA_WITH_DES_CBC_SHA"

names the same cipher suite from the default SSL provider, which in this case is IAIK.

Multiple cipher suites can be used with SSL. However, care should be taken when selecting a mechanism with multiple cipher suites. Some cipher suites have different credentials properties than others. Some only can be used with certain certificates. Say, your credentials consists of a DSA certificate, you cannot use RSA signed cipher suites. Rather than throw an exception if one cipher suite cannot be used, the SSL_IAIK Vault, which lies in the internals of the system, will eliminate any cipher suites from the list that cannot be used. However, if there is no common cipher suites that are common with the certificates given to authenticate the principal and the ones specified with the mechanism, then a **CORBA::BAD_PARAM** exception is raised.

Also, some cipher suites have different secure association properties than others. Some cipher suites only provide authentication and integrity, but not confidentiality. Others cannot authenticate a client. If a set of ciphers suites specified have different sets of association capabilities only the common association capabilities are set in the **accepting_options_supported** and the **invocation_options_supported** attributes of the **Credentials** object. Such mixing the "anon" cipher suites with and "DSS" cipher suites will not get you the ability to authenticate the client, i.e. the

EstablishTrustInClient Association Option will not be set in the created **Credentials** object's **accepting_options_supported** attribute.

If you have the ORBASEC SL2-SSL distribution, the **Current::get_supported_mechanisms** operation will return an array of **Security::MechandOptions** structures, each of which will list an "SSL_IAIK" mechanism with cipher suites that have exactly the same common association options that are supported for those cipher suites.

Also, a utility class called **MechUtil** in the **orbasec.corba** package contains static string definitions of SSL mechanisms with cipher suites grouped in a comprehensive fashion. The names are somewhat self explanatory. However, please check the JavaDoc built documentation for the exact details. The mechanism strings defined in **orbasec.corba.MechUtil** class are defined by the static constants:

- `MechUtil.SSL_NON_ANON_MECH`
- `MechUtil.SSL_NON_ANON_EXPORT_MECH`
- `MechUtil.SSL_NON_ANON_NON_EXPORT_MECH`
- `MechUtil.SSL_DH_ANON_MECH`
- `MechUtil.SSL_DH_ANON_EXPORT_MECH`
- `MechUtil.SSL_DH_ANON_NON_EXPORT_MECH`
- `MechUtil.SSL_DH_DSS_MECH`
- `MechUtil.SSL_DH_DSS_EXPORT_MECH`
- `MechUtil.SSL_DH_DSS_NON_EXPORT_MECH`
- `MechUtil.SSL_DHE_DSS_MECH`
- `MechUtil.SSL_DHE_DSS_EXPORT_MECH`
- `MechUtil.SSL_DHE_DSS_NON_EXPORT_MECH`
- `MechUtil.SSL_DH_RSA_MECH`
- `MechUtil.SSL_DH_RSA_EXPORT_MECH`
- `MechUtil.SSL_DH_RSA_NON_EXPORT_MECH`
- `MechUtil.SSL_DHE_RSA_MECH`
- `MechUtil.SSL_DHE_RSA_EXPORT_MECH`
- `MechUtil.SSL_DHE_RSA_NON_EXPORT_MECH`
- `MechUtil.SSL_RSA_MECH`
- `MechUtil.SSL_RSA_EXPORT_MECH`
- `MechUtil.SSL_RSA_NON_EXPORT_MECH`

Note – In order to use any cipher suites with RSA or RC4 in them, you are required to obtain a license from RSA, Inc. The ORBASEC SL2-SSL distribution comes with RSA disabled. In order to get ORBASEC SL2-SSL to use the RSA

cipher suites, you need to obtain from Adiron a special on-site consulting agreement to get RSA cipher suites enabled. Adiron can only do this after proof that a license from RSA has been granted.

Security Name

The **security_name** parameter must be in a **orbasec.corba.Opaque** byte encoding of a DirectoryName. This requires creating a DN. Please see your IAIK toolkit for an example of how to construct a DN:

```
import orbasec.corba.Opaque;


iaik.asn1.structures.Name my_name = //..... create a DN
byte[] security_name =
    Opaque.encodeDirectoryName(my_name).getEncoding();
```

If the value of the **security_name** parameter has the following value:

- `new byte[0]`

the principal's DN is retrieved from the certificate chain that is specified in the certificate file, which is specified in the authentication data. If the value of the **security_name** parameter is nonempty it is compared with the name in the certificate. If they do not match, a CORBA::BAD_PARAM exception is raised.

The most common use would be to leave the **security_name** parameter empty and let the principal's DN come from the certificate.

Authentication Data

The authentication data is the value of the **auth_data** parameter for the **authenticate** operation. The format for this parameter is the standard Java string to byte encoding of a Java String containing name-value pairs in the form for the **java.util.Properties** class. This format requires each entry to have the form "name=value" separated by newline characters.

For example, a call in Java to create credentials for a principal with the DN of "C=US, O=Adiron, CN=Bart" a call to the **PrincipalAuthenticator** object would be:

```
import org.omg.Security.*;
import org.omg.SecurityLevel2.*;
import orbasec.corba.Opaque;
import orbasec.corba.MechUtil;

....
    PrincipalAuthenticator pa = // get the PA from Current

    CredentialsHolder      credh = new CredentialsHolder();
    OpaqueHolder           contdata = new OpaqueHolder();
    OpaqueHolder           authspecdata = new OpaqueHodler();
    AuthenticationStatus   stat;

    stat = pa.authenticate(
                0,
                MechUtil.SSL_DH_DSS_MECH,  // mechanism
                new byte[0],               // security_name
                (“certchain=FILE:bart.dsa\n”   +
                 “password=\”MyPassword\”\n”).getBytes(),
                new SecAttribute[0],
                credh,
                contdata,
                authspecdata
    );
```

The names of the properties that are valid for the authentication data are described below.

certchain

This field names the location of X.509 Certificate chain *and* private key that you want to use. The **certchain** specification has a two part format:

```
<type>:<cert chain name>
```

Currently, the only available type is “`FILE`”. The “`FILE`” type names a file on the local system. The file must contain a certificate chain and an encrypted private key, in either DER or PEM format. Please see **iaik.utils.KeyAndCertificate** from your IAIK SSL and JCE toolkits for details.

If the **certchain** specification is empty, the **Credentials** object can only be used to set up anonymous communication.

password

This property field must contain the password for the private key. Unfortunately, we have a very minor character translator, so special characters like control characters, tabs, backspaces, and such are not representable.

Example of a creation of an Anonymous SSL Credentials Object

We have shown an example above for authenticating a principal using his certificate and encrypted private key file. Below, the following shows an example should one want to create an anonymous SSL **Credentials** object.

```
import org.omg.Security.*;
import org.omg.SecurityLevel2.*;
import orbasec.corba.MechUtil;

....
    PrincipalAuthenticator pa = // get the PA from Current

    CredentialsHolder      credsh = new CredentialsHolder();
    OpaqueHolder           contdata = new OpaqueHolder();
    OpaqueHolder           authspecdata = new OpaqueHodler();
    AuthenticationStatus   stat;

    stat = pa.authenticate(
                0,                      // auth method
                MechUtil.SSL_ANON_MECH,  // mechanism
                new byte[0],            // security_name
                new byte[0],            // auth_data
                new SecAttribute[0],    // privileges
                credsh,                 // out Credentials
                contdata,
                authspecdata
    );
```

An anonymous SSL Credentials object can be used for private and integrity based communication using Diffe-Hillman key exchange cipher suites, i.e. the cipher suites that are listed in the `MechUtil.SSL_ANON_MECH`string definition.

Authentication of IIOP Credentials

This section explains the process for creating an IIOP Credentials object. IIOP Credentials are used to identify and set up communication with standard CORBA servers and clients within ORBAsec SL2. ORBAsec SL2, by default, does not allow any insecure communication. To do so, would open up a security hole. However, there is a need for a controlled secure application to be able to communicate with insecure, standard IIOP clients and servers. An application may not communicate with an insecure, standard IIOP client or server unless it has created IIOP credentials, in the following maner.

Mechanism

The mechanism must be specified as “IIOP”.

Security Name

The security name must be specified as:

- `new byte[0]`

Authentication Data

The authenitication data can have just one optional value, “enable_server”.

enable_server

If the “`enable_server`” property is set to “`true`”, it allows insecure IIOP connections to come into your application. If this property is false, the application may only iniitate insecure IIOP connections as a client.

Example of a creation of an Anonymous SSL Credentials Object

We have shown an example above for authenticating a principal using his certificate and encrypted private key file. Below, the following shows an example should one want to create an anonymous SSL **Credentials** object.

```
import org.omg.Security.*;
import org.omg.SecurityLevel2.*;

....
```

```
PrincipalAuthenticator pa = // get the PA from Current

CredentialsHolder      credsh = new CredentialsHolder();
OpaqueHolder           contdata = new OpaqueHolder();
OpaqueHolder           authspecdata = new OpaqueHodler();
AuthenticationStatus   stat;

stat = pa.authenticate(
            0,                          // auth method
            “IIOP”,                     // mechanism
            new byte[0],                // security_name
            (“enable_server=true\n”).getBytes(),
                                        // auth_data
            new SecAttribute[0],        // privileges
            credsh,                     // out Credentials
            contdata,
            authspecdata
);
```

An “anonymous” IIOP Credentials object can be used for insecure communication as both a client and a server. The security attributes of the IIOP Credentials object will tell of the local hostname and the local TCP/IP port number (if “`enable_server`” is set to “`true`”) given to the Credentials.

CHAPTER 6

Credentials

What are Credentials?

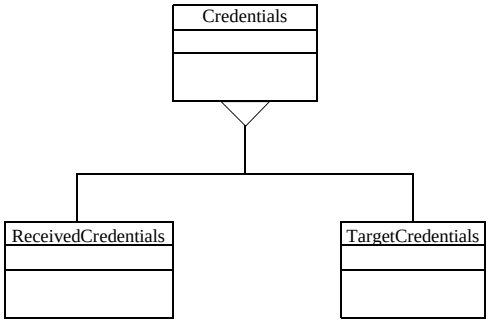
Credentials are the application programmer's interface to querying of security related attributes belonging to the application itself and of any clients making invocations. Also, one may examine the Credentials of a server. Credentials come in three flavors, "own" credentials, "received", and "target" credentials.

The "own" type of credentials represent the application's credentials from which a special authentication procedure had to be performed. Own credentials are created by making a request on the **PrincipalAuthenticator** object that resides as an attribute on the **Current** object. The principal authenticator goes through the necessary procedures to authenticate the intended security name under the intended security mechanism and requested privileges to produce a **Credentials** object that represents a principal. Own credentials are specific to the capsule, (i.e. they are not thread specific).

The "received" type of credentials are only valid in the context of servicing a request as a server object. They represent the establishment of a security context between the client and the target. The target object can query the "received" credentials object to identify the principal making the request, and query any special privileges that the principal may have acquired. Received credentials are specific to the execution context in servicing a request, (i.e. they are thread specific).

The “target” type of credentials are the credentials of an object behind the object reference. It may be desirable to examine that an object has the right credentials before you start making requests on it. These type of credentials are for examination only. They cannot be used to make invocations like “own” and “received” credentials can

FIGURE 1. The Credentials Interfaces.



The next two sections explain the **Credentials** interface, the **ReceivedCredentials** interface, and the **TargetCredentials** interface. The **Credentials** interface is the base interface and is used to represent “own” credentials. A **ReceivedCredentials** object represents the security context between the client and target from the target’s point of view. A **TargetCredentials** object represents the security context between the client and target from the client’s point of view. Each of the **ReceivedCredentials** and **TargetCredentials** objects hold more information than an own credentials object.

Credentials

The **Credentials** interface is the base type for own credentials, received credentials, and target credentials, own credentials being the **Credentials** interface itself.

The **Credentials** interface holds information pertaining to the authenticated identity of the subject of the credentials, i.e. the principal. **Credentials** are a Security Level 2 module interface. However, the implementation is dependent on the underlying security mechanisms that are installed. The **Vault,** a Security Replaceable

object, creates **Credentials** objects. A **Credentials** object is specific to the security mechanisms supported by that **Vault**. The **Credentials** interface has the following definition:

```
// IDL
interface Credentials { // Locality Constrained
  Credentials copy();

  void destroy():

  readonly attribute Security::CredentialsType
                                credentials_type;

  readonly attribute Security::AuthenticationState
                                authentication_state;

  readonly attribute Security::MechanismType  mechanism;

  attribute Security::AssociationOptions
                                accepting_options_supported;
  attribute Security::AssociationOptions
                                accepting_options_required;
  attribute Security::AssociationOptions
                                invocation_options_supported;
  attribute Security::AssociationOptions
                                invocation_options_required;
  boolean get_security_feature(
    in   Security::CommunicationDirection   direction,
    in   Security::SecurityFeature          feature
  );
```

```
  boolean set_attributes (
    in  Security::AttributeList       requested_attributes,
    out Security::AttributeList       actual_attributes
  );

  Security::AttributeList get_attributes(
    in  Security::AttributeTypeList   attributes
  );

  boolean is_valid (
    out Security::UtcT                expiry_time
  );

  boolean refresh(
    in  Security::Opaque              refresh_data
  );
};
```

The attributes and operations of the **Credentials** object's interface are:

copy

This operation is produces a "deep" copy of the **Credentials** object. There are semantic issues with what this operation means in the context of the **destroy** operation. These issues have not yet been resolved. Guidelines for the implementation of this method are presenting in the section on "The Vault" on page 144.

The **copy** operation's interface is below:

```
// IDL
Credentials copy();

// Java
public org.omg.SecurityLevel2.Credentials copy();
```

destroy

This operation is destroys the copy of the **Credentials** object.

The destroy operation's interface is below:

```
// IDL
void destroy();

// Java
public void destroy();
```

credentials_type

This attribute contains the value discerning whether the credentials are of the “own”, “received”, or “target” type.

```
// IDL
readonly attribute Security::CredentialsType
                                                credentials_type;
// Java
public org.omg.Security.CredentialsType
credentials_type();
```

This operation returns **SecOwnCredentials** if the **Credentials** is of the “own” credentials type. It returns **SecReceivedCredentials** if the **Credentials** object is of the “received” credentials type and can be narrowed to a **ReceivedCredentials** object. It returns **SecTargetCredentials** if the **Credentials** object is of the “target” credentials type and can be narrowed to a **TargetCredentials** object.

authentication_state

Since **Credentials** objects may take several operations to fully become initialized this read-only attributes serves as an indication of the authentication state, which is the same as the result returned from **PrincipalAuthenticator::authenticate** and **PrincipalAuthenticator::continue_authentication** operations.

```
// IDL
readonly attribute Security::AuthenticationStatus
                                          authentication_state;
// Java
public org.omg.Security.Authenticationstatus
authentication_state();
```

This attribute has the value of **SecAuthSuccess** if the **Credentials** are fully initialized. It returns **SecAuthContinue** if subsequent calls to **PrincipalAuthenticator::continue_authentication** are needed. It returns **SecAuthFailure** if the

continuing authentication of the **Credentials** has failed. It returns **SecAuthExpired** if the continuing authentication of the **Credentials** is no longer viable.

In both the ORBASEC SL2-GSSKRB and ORBASEC SL2-SSL distributions, the default authentication method is a one step process, and therefore the **PrincipalAuthenticator** object only creates **Credentials** with **SecAuthSuccess** for an authentication state. Should the call to **PrincipalAuthenticator::authenticate** fail, a **Credentials** object is not created.

mechanism

This read only attribute specifies the symbolic name security mechanism and the symbolic name of the cipher suites that the credentials support.

```
// IDL
readonly attribute Security::MechanismType mechanism;

// Java
public String mechanism();
```

Please see the section on “Mechanism” on page 89. for detail.

accepting_options_supported

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the server side. It also serves as the value that is placed in the “target_supports” field of the security component (should one exist) for the particular security mechanism in an objects’s IOR.

Setting of the attribute’s value to an illegal set of **Security::AssociationOptions** raises a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    accepting_options_supported;
// Java
public short accepting_options_supported();
public void accepting_options_supproted(short opts);
```

Accepting options supported must be non-zero to be used with **SecLev2::Current::set_accepting_credentials** operation. The absolute minimum in security

terms that any credentials object can have in supported options to establish an association is:

NoProtection + NoDelegation

Note – Only “own” credentials will have accepting options that are not zero. This attribute having a value of zero simply states that this credentials object cannot be used to establish secure associations on the server side. A “received” credentials object will have accepting options of zero. A “target” credentials object will have a value of zero.

After Credentials are fully initialized the user can change the options these credentials support. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the server side. They cannot be set to less than the **accepting_options_required** attribute. If one must decrement the options that are supported, one must set the required options first. The options that are supported cannot be set to more than the options that the credentials were created with. Credentials are created with their maximum supported options set in this attribute.

In the ORBASEC SL2-GSSKRB distribution Kerberos credentials initially support the following association options on the server side:

NoProtection, Integrity, Confidentiality, Detect Replay, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation,SimpleDelegation.

The user may not set them less than NoProtection, NoDelegation.

In the ORBASEC SL2-SSL distribution, the options supported for SSL **Credentials** objects depend on the cipher suites that were specified in the **PrincipalAuthenticator::authenticate** operation. Most cipher suites have the following options set:

NoProtection, Integrity, Confidentiality, Detect Replay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation

However, anonymous based cipher suites leave out EstablishTrustInClient and EstablishTrustInTarget. Some DH cipher suites do not encrypt, and therefore leave out Confidentiality. The listed according to the SSL mechanism defined in **orbasec.corba.MechUtil** are as follows:

Mechanism	Association Options Supported
MechUtil.SSL_DH_ANON_MECH	Integrity, DetectReplay, DetectMisordering, NoDelegation
MechUtil.SSL_DH_DSS_MECH	Integrity, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_DH_RSA_MECH	Integrity, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_DHE_DSS_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_DHE_RSA_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_RSA_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_NON_ANON_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation

TABLE 4. SSL Cipher Suite Accepting Options Supported

accepting_options_required

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the server side. It also serves as the value that is placed in the “target_requires” field of the security component (should one exist) for the particular security mechanism in an objects’s IOR.

Setting of the attribute’s value to an illegal set of **Security::AssociationOptions** raises a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    accepting_options_required;
// Java
public short accepting_options_required();
public void accepting_options_required(short opts);
```

Accepting options required may be zero.

After Credentials are fully initialized the user can change the options these credentials require. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the server side. They cannot be set to more than the **accepting_options_supported** attribute. If one must augment the options that are required, one must set the supported options first.

In the ORBASEC SL-GSSKRB distribution, Kerberos credentials initially have required options of zero. However, certain combinations that do not make sense are illegal to be set, such as, you cannot set NoProtection with any of Integrity, Confidentiality, or Detect Replay. Likewise, you cannot set both NoDelegation and SimpleDelegation to be required.

In the ORBASEC SL2-SSL distribution, the options that can be set to be required follow the same restrictions.

invocation_options_supported

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the client side.

Setting of the attribute's value to an illegal set of **Security::AssociationOptions** raises a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    invocation_options_supported;
// Java
public short invocation_options_supported();
public void invocation_options_supported(short opts);
```

Invocation options supported must be non-zero to be used with an **SecurityLevel2::InvocationCredentialsPolicy**. The absolute minimum in security terms that any credentials object can have in supported options to establish an association is:

NoProtection + NoDelegation

Note – In the case of delegation, "received" credentials may have supported invocation options. Having a value of zero simply states that this credentials object cannot be used to establish secure associations on the client side. A "target" credentials object will have a value of zero.

After Credentials are fully initialized the user can change the options these credentials support. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the client side. They cannot be set to less than the **invocation_options_required** attribute. If one must decrement the options that are supported, one must set the required options first. The options that are supported cannot be set to more than the options that the credentials were created with. Credentials are created with their maximum supported options set in this attribute.

In the ORBASEC SL2-GSSKRB distribution Kerberos credentials initially support the following association options on the server side:

NoProtection, Integrity, Confidentiality, Detect Replay, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation,SimpleDelegation.

The user may not set them less than NoProtection, NoDelegation.

In the ORBASEC SL2-SSL distribution, the options supported for SSL **Credentials** objects depends on the cipher suites that were specified in the **PrincipalAuthenticator::authenticate** operation. Most cipher suites have this set:

NoProtection, Integrity, Confidentiality, Detect Replay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation

However, anonymous based cipher suites leave out EstablishTrustInClient and EstablishTrustInTarget. Some DH cipher suites do not encrypt, and therefore they leave out Confidentiality. The list according to the SSL mechanism defined in **orbasec.corba.MechUtil** class are as follows:

Mechanism	Association Options Supported
MechUtil.SSL_DH_ANON_MECH	Integrity, DetectReplay, DetectMisordering, NoDelegation
MechUtil.SSL_DH_DSS_MECH	Integrity, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_DH_RSA_MECH	Integrity, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation

TABLE 5. SSL Cipher Suite Invocation Options Supported

Mechanism	Association Options Supported
MechUtil.SSL_DHE_DSS_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_DHE_RSA_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_RSA_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_NON_ANON_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation

TABLE 5. SSL Cipher Suite Invocation Options Supported

invocation_options_required

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the client side.

Setting of the attribute's value to an illegal set of **Security::AssociationOptions** raises a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    invocation_options_required;
// Java
public short invocation_options_required();
public void invocation_options_required(short opts);
```

Invocation options required may be zero.

After Credentials are fully initialized the user can change the options these credentials require. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the client side. They cannot be set to more than the **invocation_options_supported** attribute. If one must augment options that are required, one must set the supported options first.

In the ORBASEC SL-GSSKRB distribution, Kerberos credentials initially have required options of zero. However, certain combinations that do not make sense are illegal to be set, such as, you cannot set NoProtection with any of Integrity, Confi-

dentiality, or Detect Replay. Likewise, you cannot set both NoDelegation and SimpleDelegation to be required.

In the ORBASEC SL2-SSL distribution, the options that can be set to be required follow the same restrictions.

get_security_feature

This operation returns a boolean that represent the feature state of the credentials.

```
// IDL
boolean get_security_feature(
    in Security::CommunicationDirection   direction,
    in Security::SecurityFeature          feature
);

// Java
public boolean get_security_features(
    int                               direction,
    org.omg.Security.SecurityFeature feature
);
```

set_attributes

This operation is intended for use in attribute management of the particular credentials. Its meaning is defined to diminish attributes of the credentials in the context of the mechanism's ability. It may be desirable to diminish the set of attributes that a Credentials object contains. No all mechanisms can support this operation. Depending on the mechanism, some attributes may not be removed.

The **set_attributes** operation's interface is below:

```
// IDL
boolean set_attributes(
  in   Security::AttributeList   requested_attributes,
  out  Security::AttributeList   actual_attributes
);

// Java
public boolean set_attributes(
    org.omg.Security.SecAttribute[]      requested_attributes,
    org.omg.Security.AttributeListHolder actual_attributes
);
```

The value given to the **requested_attributes** parameter must be a subset of the list of attributes returned from the **get_attributes** operation. If it contains an attribute not from that the list of attributes from the **get_attributes** operation, a **CORBA::BAD_PARAM** exception is raised. The value returned in the **actual_attributes** parameter is the resultant list of all the attributes the Credentials object now contains. The return value returns true if the operation was successful and the actual attributes are indeed the requested attributes. If return value of the operation is false (i.e. no exception is raised), the operation is considered successful, however, some attributes in the Credentials object that were not given to the **requested_attributes** parameter were not removed.

Note – This operation is not effectively supported by the ORBASEC SL2-GSSKRB or ORBASEC SL2-SSL distributions as the implementations of the Kerberos and SSL protocols have minimal attributes that cannot be removed.

get_attributes

This operation returns an unordered sequence of security attributes that belong to the credentials.

```
// IDL
Security::AttributeList get_attributes(
  in   Security::AttributeTypeList    attributes
);

// Java
public org.omg.Security.SecAttribute[] get_attributes(
     org.omg.Security.AttributeType[]  attributes
);
```

Security attributes come in many types and values. Please see the section on Security Attributes for further details.

Although there is a standard for the attribute types and the values to which they refer, no standardization effort is underway to define the format of the values of the particular attributes.

is_valid

This operation returns a boolean value indicating whether the credentials are still valid. The output parameter returns the time of expiration.

```
// IDL
boolean is_valid(
  out   Security::UtcT expiry_time
);

// Java
public boolean is_valid(
    org.omg.TimeBase.UtcTHolder expiry_time
)
```

refresh

This operation is intended to renew a credentials before it may expire. It returns a boolean value indicating the success of the renewal.

```
// IDL
boolean refresh(
   in  Security::Opaque  refresh_data
);

// Java
public boolean refresh( byte[] refresh_data );
```

In the ORBASEC SL2-GSSKRB distribution, this operation is supported for Kerberos credentials of the "own" type only. If invoked on **Credentials** of the "received" or "target" type it raises a **CORBA::BAD_OPERATION** exception. If invoked on **Credentials** of the "own" type, it returns true if the operation succeeds, however, it raises an exception with an informative error message if the operation fails.

Note – For the current version of the GSS-Kerberos mechanism credentials, the **refresh_data** is required to be octet sequence of zero length.

In the ORBASEC SL2-SSL distribution, this operation is not supported for SSL credentials. If invoked it raises a **CORBA::BAD_OPERATION** exception.

Received Credentials

On the target side a **ReceivedCredentials** object represents a secure association between the client and target. Received credentials must have more information than “own” credentials.

The interface inherits from the **Credentials** interface, and in the case of using the received credentials for invocations, the invocation features, operations, and attributes of the **Credentials** object have the same meaning. Of course, the **credentials_type** attribute is set to **SecReceivedCredentials**. Its interface is defined below:

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

accepting_credentials

This read-only attribute is the **Credentials** objects used to establish the secure association with the client.

```
// IDL
readonly attribute Credentials accepting_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
accepting_credentials();
```

association_options_used

This read-only attribute states the association options that were used to make the association with the client using the **accepting_credentials**. This value should be a value somewhere between the **accepting_options_required** and the **accepting_options_supported** of the **accepting_credentials**.

```
// IDL
readonly attribute Security::AssociationOptions
                                        association_options_used;
// Java
public short association_options_used();
```

delegation_state

This read-only attribute is the value of the delegation state of the *client's own credentials*. It states whether the immediate invoking principal of the operation is the initiator or a delegate of some other principal.

```
// IDL
readonly attribute Security::DelegationState delegation_state;

// Java
public org.omg.Security.DelegationState
delegation_state();
```

Note – For some security mechanisms, this information is indeterminable. When this information is indeterminable, impersonation is assumed; and therefore, this attribute has the value of **SecInitiator**.

In the ORBASEC SL2-GSSKRB distribution, only unrestricted or simple delegation is supported for Kerberos credentials. Therefore, Kerberos credentials that are received have the value of this attribute set to **SecInitiator**, since the Kerberos protocol cannot determine the delegation state of the client.

In the ORBASEC SL2-SSL distribution, no form of delegation is supported so this attribute always has the value of **SecInitiator**.

delegation_mode

This read-only attribute states the delegation mode of the received credentials. It stipulates that the credentials are in the a delegation mode of:

- No delegation mode (**SecDelModeNoDelegation**), where they can not be used for invocations.
- Simple delegation mode (**SecDelModeSimpleDegation**), where the credentials can be indiscriminately used on the client's behalf.

- Composite delegation (**SecDelModeCompositeDelegation**) where the credentials have some sort of composite ability, such as a trace, a combination of privileges, etc.

```
// IDL
readonly attribute Security::DelegationMode delegation_mode;

// Java
public org.omg.Security.DelegationMode
delegation_mode();
```

In the ORBASEC SL2-GSSKRB distribution, Kerberos credentials support no delegation and simple delegation, but not composite delegation. In the ORBASEC SL2-SSL distribution, SSL credentials do not support any form of delegation.

Target Credentials

On the client side a **TargetCredentials** object represents a secure association between the client and target. Target credentials must have more information than "own" credentials.

The interface inherits from the **Credentials** interface. The **TargetCredentials** object cannot be used for invocations. The **credentials_type** attribute is set to **SecTargetCredentials**. Its interface is defined below:

```
interface TargetCredentials : Credentials {
                                    // Locality Constrained
  readonly attribute Credentials     initiating_credentials;
  readonly attribute Security::AssociationOptions
                                       association_options_used;
};
```

initiating_credentials

This read-only attribute is the **Credentials** objects used to establish the secure association with the server.

```
// IDL
readonly attribute Credentials initiating_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
initiating_credentials();
```

association_options_used

This read-only attribute states the association options that were used to make the association with the target using the **initiating_credentials**. This value should be a value somewhere between the **accepting_options_required** and the **accepting_options_supported** of the **initiating_credentials**.

Security Attributes of Credentials

Security attributes are used to represent the characteristics of the principal behind the Credentials object. They are defined by the following IDL.

```
// IDL
struct ExtensibleFamily {
    unsigned short  family_definer;
    unsigned short  family;
};

typedef unsigned long SecurityAttributeType;

struct AttributeType {
    ExtensibleFamily        attribute_family;
    SecurityAttributeType   attribute_type;
};

struct SecAttribute {
    AttributeType           attribute_type;
    Opaque                  defining_authority;
    Opaque                  value;
};
```

Security attributes come in many types and have many different values. There is not yet a clear standard for defining the types and values of security attributes. The

OMG has defined several attribute type values, but does not yet define there value types. However, a standard mechanism for defining security attributes (i.e. their families, types, and values) exists.

CORBA Family of Security Attributes

The OMG defines security attributes by the AttributeType structure. The Attribute-Type structure is parameterized with a family type. That family type is defined by an authority. The **family_definer** field of the **ExtensibleFamily** structure indicates the authority that defined the attribute. This tag is registered with the OMG. The OMG reserves a family definer value of zero for CORBA.:

TABLE 6. CORBA Family Definer

CORBA Family Definer
0

CORBA currently defines two families of attributes.

TABLE 7. CORBA Families

CORBA Family	Description
0	Identity
1	Privileges

CORBA also defines a number of constants for the **attribute_type** field of the **AttributeType** structure. These constants are defined in the Security module of the CORBA Security Specification and are not listed here. Unfortunately there are no standards for the **defining_authority** and **value** fields for attributes of these types.

Adiron Family of Security Attributes

Adiron uses the OMG mechanism for defining its own security attributes for ORBASEC SL2. This procedure involves creating an families of attributes. A fam-

ily is defined by an authority. In this case, Adiron is the authority. Adiron registers a family definer tag by the OMG. It is below:

TABLE 8. Adiron Family Definer

Adiron Family Definer
0xA11C

AttributeType

ORBASEC SL2 uses several of its own types. These types are in families defined by Adiron's family definer, 0xA11C (41244 decimal). Adiron currently defines the following attribute families:

TABLE 9. Adiron Families

Family Definer	Family	Description
0xA11C	0	Miscellaneous
	1	Internet
	2	Identity

Adiron also defines the following attribute types.

TABLE 10. Adiron Security Attribute Types

Adiron Family	Security Attribute Type	Description
0	0	Security Mechanism of Credentials
1	1	Local Host Address
	2	Local Port Number
	3	Peer Host Address
	4	Peer Port Number
2	1	Subject Identifier
	2	Issuer Identifier

Defining Authority

Adiron uses one value for the defining_authority attribute for all its attributes. It is an Opaque encoding (See chapter on "Opaque Encodings" on page 185) of the printable string "Adiron".

TABLE 11. Adiron Defining Authority

Adiron Defining Authority Value
`Opaque.encodePrintableString("Adiron").getEncoding()`

Value

The values of the Adiron security attributes use the Opaque encoding scheme (See chapter on "Opaque Encodings" on page 185).

TABLE 12. Adiron Attribute Values

Attribute Type	Value Description
Security Mechanism Type	This value is an Opaque encoding of a PrintableString. The string that is encoded is the mechanism type of the credentials.
Local Host Address	This value is an Opaque encoding of an IPAddress (octet sequence). This IP address of the local machine.
Local Port Number	This value is an Opaque encoding of an IPPortNumber (integer). This IP port number is that of the local machine.
Peer Host Address	This value is an Opaque encoding of an IPAddress (octet sequence). This IP address is that of the remote host. This attribute only exists in ReceivedCredentials or TargetCredentials.
Peer Port Number	This value is an Opaque encoding of an IPPortNumber (integer). This IP port number is that of the remote machine. This attribute only exists in ReceivedCredentials or TargetCredentials.
Subject Identity	This value is an Opaque encoding of a value specific to the mechanism (see below). It is the identify attribute of the principal.
Issuer Identity	This value is an Opaque encoding of a value specific to the mechanism (see below). It is the identity attribute of the principal that vouches for the subject principal.

If you have the ORBASEC SL2-GSSKRB distribution, the **value** field of the Subject Identity and the Issuer Identity attributes contain the Opaque encoding of a KerberosName that is the principal's name, such as "bart@MYREALM.COM". The **defining_authority** field contains the name of the ticket granting ticket service for that realm as a name encoding of a KerberosName, such as "krbtgt/MYREALM.COM@MYREALM.COM".

If you have the ORBASEC SL2-SSL distribution, the **value** field of the Subject Identity and the Issuer Identity attributes contain the Opaque encoding of a DirectoryName that is the principal's name, which was found in the SubjectDN or IssuerDN fields of the prnicipal's X.509 certificate. This value, in its raw form is a DER encoding of an ASN.1 DN. A string representation of such a structure might be "C=US, O=Adiron, OU=R&D, CN=Bart". If using anonymous ciphers, the **value** field will contain the name encoding of a PrintableString containing "anonymous".

CORBA Family 1 AccessId

The **AccessId** is defined by CORBA Family 1, and its attribute type identifier is 2. The ORBASEC SL2-GSSKRB and SL2-SSL both create this attribute in the following manner:

If you have the ORBASEC SL2-GSSKRB distribution, the **value** field of the **AccessId** attribute contains the Opaque encoding of a KerberosName that is the principal's name, such as "bart@MYREALM.COM". The **defining_authority** field contains the name of the ticket granting ticket service for that realm as a name encoding of a KerberosName, such as "krbtgt/MYREALM.COM@MYREALM.COM".

If you have the ORBASEC SL2-SSL distribution, the **value** field contains the Opaque encoding of a DirectoryName that is the principal's name, which was found in the SubjectDN field of the prnicipal's X.509 certificate. This value, in its raw form is a DER encoding of an ASN.1 DN. A string representation of such a structure might be "C=US, O=Adiron, OU=R&D, CN=Bart". The **defining_authority** field contains the name of immediate issuer, which is the certificate authority that issued the principal's X.509 certificate. It comes directly from the IssuerDN field of the principal's X.509 certificate. If using anonymous ciphers, both attribute fields will contain the name encoding of a PrintableString containing the string "anonymous".

How are the Credentials Related to the IOR?

The list of own type credentials represents the information that is placed in the tagged components section of the **IIOP 1.1** profile of the IOR.

Each **Credentials** object that comes from the set of designated accepting credentials [see “Accepting Credentials Attributes and Operations” on page 71] places a security component representing its capabilities and security name in the IIOP profile.

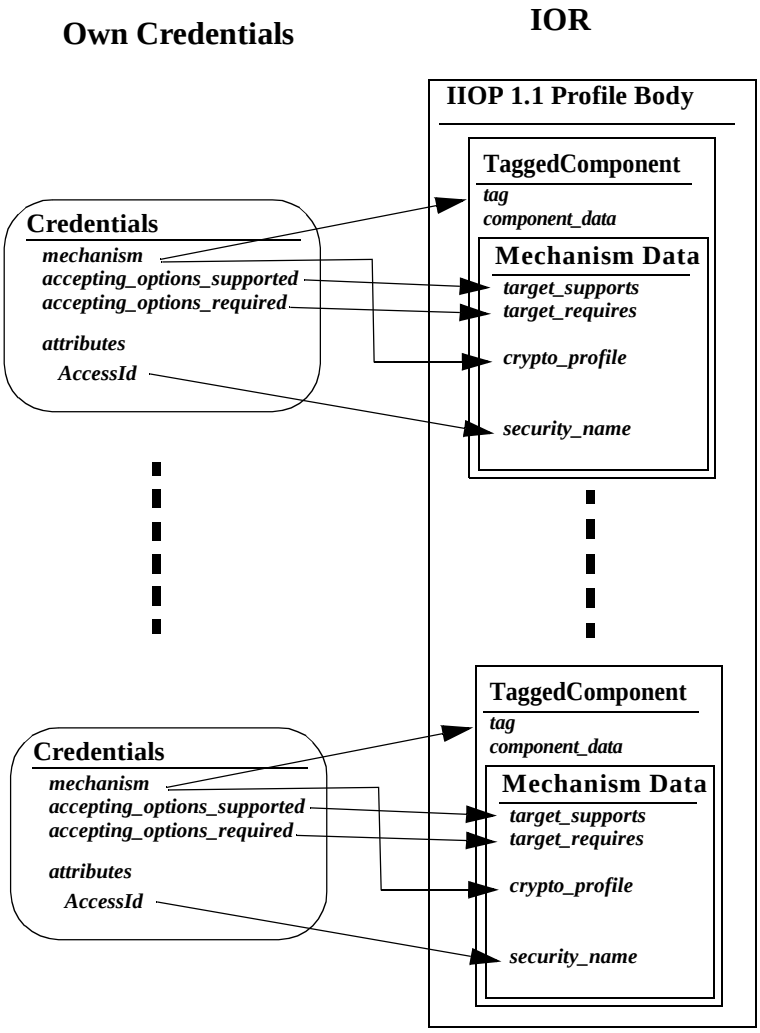


FIGURE 2. Mapping of Own Credentials Objects to IOR

When the **object_to_string** operation on the ORB is called on an object, or an object reference is given to a client via a return value or an output parameter an IOR is created for the object. ORBASEC SL2 adds a security component for each valid accepting **Credentials** object.

A tagged security component in general has the following format:

```
//IDL
typedef unsigned long ComponentId;
typedef struct TaggedComponent {
   ComponentId     tag;
   sequence<octet> component_data;
};

typedef sequence<TaggedComponent> MultiComponentProfile;
```

Each security component in the IOR contains a **tag** specifying the mechanism, and a **component_data** attribute that contains the mechanism data. The structure for most mechanism data has the same format (except for SSL), illustrated below with the **KerberosV5** structure. It is not represented by a common type, because mechanisms of the future may require extended information.

```
//IDL
module SECIOP {
typedef sequence<octet>            SecurityName;
typedef short                      CryptographicProfile;
typedef sequence<CryptographicProfile>
                                   CryptographicProfileList;

// Protocol Component for SECIOP
struct SECIOP_INET_SEC_TRANS {
    unsigned short        port;
};

// component_data attribute of a TaggedComponent.
struct KerberosV5 {
    Security::AssociationOptions    target_supports;
    Security::AssociationOptions    target_requires;
    CryptographicProfileList        crypto_profile;
    SecurityName                    security_name;
};
};
```

The **Credentials** object's **mechanism** attribute contains a combination of the security component tag and the cryptographic profiles that the mechanism supports in string form. The string has the form of the integer tag of the mechanism, i.e. 17 for KerberosV5, and separated by a comma, numbers only relevant to that mechanism,

i.e. 11 represents the DES-CBC-MD5 cryptographic profile for the KerberosV5 mechanism. [4, Section A.11.4 Security Mechanisms]

For example, the value of the **Credentials** object's **mechanism** attribute of `"Kerberos,DES-CBC-MD5"` will create the **IOP::TaggedComponent** with a **tag** of 17 and a **component_data** field containing the encapsulated value of the **KerberosV5** structure.

The numbers trailing the first number in the **mechanism** attribute are the cryptographic profile numbers, which are also comma separated. These numbers are directly mapped to a sequence of short values that are placed in the **crypto_profile** attribute of the mechanism data. The utility class **orbasec.corba.MechUtil** has these number to symbolic cryptographic profiles associations.

An application programmer controls the capabilities advertised in the IOR by manipulating the **Credentials** object's **accepting_options_supported** and **accepting_options_required** attributes. The values of these attributes are mapped directly to the **target_supports** and **target_requires** attributes of the security component.

The **security_name** attribute is the value of the **AccessId** typed security attribute of the **Credentials**. At the API, the security name as a value parameter of the AccessId security attribute is a **orbasec.corba.Opaque** encoding. However, for the IOR its "raw" byte encoding, is the one that is placed in this field.

In the ORBASEC SL2-GSSKRB distribution the security name is the octet sequence directly mapped to an ASCII string containing the Kerberos principal name fully qualified with the realm name, i.e. `name@REALM`, or `name/instance@REALM`.

Important Temporal Considerations

One must be cautious as to the times at which **Credentials** object's accepting options are modified and the times when object references are given out or converted to strings using the ORB operation **object_to_string**.

Once an IOR is created for an object reference it contains a snapshot of the state of the credentials. If the application programmer modifies the credentials accepting options after object references are given out, then those objects references may be rendered ineffective. They no longer represent the current security state of the object to which they are referring.

Extensions for ORBASEC SL2-SSL Credentials

The CORBA security credentials model is insufficient for examining the some aspects of X509 certificate chains. ORBASEC SL2-SSL does verify that every certificate in the chain verifies with the public key of its issuer, or is in line with the TrustedAuthorityPolicy. See "TrustedAuthorityPolicy" on page 138. ORBASEC SL2-SSL also verifies that the certificate is still valid with respect to the current system time. However, for those who need to analyze the certificates with a bit more fervor, you can get at the certificate chain on the credentials object by *casting* the **org.omg.SecurityLevel2.Credentials** object to an **orbasec.ssliop.iaik.Credentials** object and use its **certificate_chain** method to retrieve the certificate chain associated with the credentials object. An example follows:

```
// Java
org.omg.SecurityLevel2.Credentials rcreds =
                                        current.received_credentials();
java.security.cert.X509Certificate[] cert_chain =
       ((orbasec.ssliop.iaik.Credentials)rcreds).certificate_chain();
```

Since IAIK is the provider (the certificate mechanism will start with "SSL_IAIK"), more information beyond that of a **java.security.cert.X509Certificate** can be retrieved by casting to an **iaik.X509.X509Certificate**, such as follows:

```
iaik.X509.X509Certificate cert =
                              (iaik.X509.X509Certficate) cert_chain[0];
```

Please see your IAIK documentation for details on using this class.

CHAPTER 7

Policies

Policies

This section explains the various security related policies that the security service understands and that can be placed on object references. These policies can also be set as defaults for the thread by using the **set_overrides** operation on the **Current** object. This section also explains the analysis and decision procedure taken on policies to discover the parameters of a secure association with the target. The set of default policies *out-of-the-box* are presented at the end of the section.

The policies that the security service machinery understands is the following policies:

- **MechanismPolicy**
- **InvocationCredentialsPolicy**
- **DelegationDirectivePolicy**
- **QOPPolicy**
- **EstablishTrustPolicy**

All of the above policy interfaces are members of the **SecurityLevel2** module.

One policy of each type may be placed on an object reference by using the objects pseudo operation, **set_policy_overrides**.

The **orbasec.SL2** static class has factory operation that create simple policies regarding each of the above listed policies. However, that does not preclude an application developer from creating a policy object of his own device incorporating creatively produced results. For example, one may create a **QOPPolicy** that returns different **Security::QOP** values depending on the time of day, location, or other environmental considerations.

Temporal Considerations

Policy objects in ORBASEC SL2 are queried at the time a connection to a remote operation is made. The policies in place for the connection are in place for the duration of the connection.

MechanismPolicy

An object of the **SecurityLevel2::MechanismPolicy** interface specifies a set of security mechanisms from which to consider when making invocations. Its only attribute is a list of mechanism types that should be considered in order while trying to find compatible client credentials and mechanisms of the target. Please see the **PrincipalAuthenticator** section "Mechanism" on page 82 for an explanation of mechanism type identifiers.

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

// Java
package org.omg.SecurityLevel2;
public interface MechanismPolicy
            extends org.omg.CORBA.Policy
{
  String[] mechanisms();
}
```

Default Mechanism Policy

ORBASEC SL2 comes with a default mechanism policy that is set on the initial thread of execution and is inherited from every descendant's thread until it is explicitly set. The default MechanismPolicy that is to match the mechanisms of the received (should one exist) and own credentials objects.

This policy serves as an attempt to use the current credentials that have been created by the application, without having the application writer to have to think about policy objects.

The semantics of this "dynamic" mechanism policy roughly follows the implementation below:

```
// Java
package orbasec.seclev2;
import org.omg.Security.*;
import org.omg.SecurityLeve2.*;

public class DynMechansimPolicy
  extends    orbasec.corba.LocalObject,
  implements org.omg.SecurityLevel2.MechanismPolicy
{
        .....
  public String[]
  mechanisms()
  {
    // Get the invocation credentials policy
    InvocationCredentialsPolicy invocp =
        InvocationCredentailsPolicyHelper.narrow(
             current.get_policy(
                  SecInvocationCredentialsPolicy.value);
    // Create an array of strings of each credentials
    // mechanism.
    Credentials[] creds = invocp.creds();
    Vector mechs = new Vector();
    for(int i; i < invoc; i++) {
      if(creds[i].invocation_options_supported() != 0)
        mechs.addElement(creds[i].mechanism);
    String[] ms = new String[mechs.size()];
    mechs.copyInto(ms);
    return ms;
}
```

Invocation Credentials Policy

An object of the **SecurityLevel2::InvocationCredentialsPolicy** interface specifies a set of **Credentials** objects from which to consider when making invocations. Its only attribute is a list of **Credentials** objects that should be considered.

```
//IDL
interface InvocationCredentialsPolicy : CORBA::Policy {
                                        // Locality Constrained
    readonly attribute SecurityLevel2::CredentialsList creds;
};

// Java
package org.omg.SecurityLevel2;
public interface InvocationCredentialsPolicy
     extends org.omg.CORBA.Policy
{
  org.omg.SecurityLevel2.Credentials[] creds():
}
```

Default Invocation Credentials Policy

ORBASEC SL2 comes with a default invocation credentials policy. This policy dynamically selects the received credentials (if its delegation mode is not one of **SecDelModeNoDelegation**), and the own credentials list from the **Current** object. This policy serves as the default to give the application writer the default behavior of using the credentials objects he authenticates.

The semantics of this “dynamic” invocation credentials policy roughly follows the implementation below:

```
// Java
package orbasec.seclev2;
import Security.*;
import SecurityLevel2.*;

public class DynRecvOwnCredentialsPolicy
  implements orbasec.corba.LocalObject,
             InvocationCredentialsPolicy
{
 public Credentials[]
 creds()
 {
  Vector v = new Vector();
  try {
    ReceivedCredentials rcreds =
                          current.received_credentails();
    if(rcreds.accepting_options_supported() != 0)
       v.addElement(rcreds);
  } catch (BAD_OPERATION e) {
  }
  Crededentails[] own = current.own_credentials();
  for(int i = 0; i < own.length; i++ ) {
     if(own[i].invocation_options_supported != 0)
       v.addElement(own[i]);
  }
  Credentials[] creds = new Credentials[v.size()];
  v.copyInto(creds);
  return creds;
 }
}
```

QOP Policy

An object of the **SecurityLevel2::QOPPolicy** interface specifies the quality of protection that should be used when making an invocation on the target.

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

// Java
package org.omg.SecurityLeve2;
public interface QOPPolicy
    extends org.omg.CORBA.Policy
{
  public org.omg.Security.QOP qop();
}
```

Default QOP Policy

ORBASEC SL2 comes with a default QOP policy that is set on the initial thread of execution and is inherited from every descendant thread until it is explicitly set. The default QOPPolicy returns a QOP to match the invocation options that are required or supported by the credentials on the thread based Invocation Credentials Policy.

This policy serves as an attempt to use the current credentials that have been created by the application, without having the application writer to have to think about policy objects.

The semantics of this “dynamic” QOP policy roughly follows the implementation below:

```
// Java
package orbasec.seclev2;
import org.omg.Security.*;
import org.omg.SecurityLeve2.*;

public class DynQOPPolicy
  extends    orbasec.corba.LocalObject,
  implements org.omg.SecurityLevel2.QOPPolicy
{
        .....
  private QOP getQOP(short association_options)
  {
    // definition of function that translates the association
    // options to a QOP, with preœdence to Integ and Conf,
    // Conf or Integ, then NoProtection.
  }
  // Policy Function
  public QOP
  qop()
  {
    // Get the invocation credentials policy
    InvocationCredentialsPolicy invocp =
        InvocationCredentialsPolicyHelper.narrow(
             current.get_policy(
                  SecInvocationCredentialsPolicy.value);
    // Create an array of strings of each credentials
    // mechanism.
    Credentials[] creds = invocp.creds();
    QOP qop = QOP.SecQOPIntegrityAndConfientiality;
    int qopmask = NoProtection.value | Integrity.value |
                  Confidentiality.value;
    for(int i =0 ; i < creds.length; i++) {
      // Can we even use the credentials?
      if(creds[i].invocation_options_supported() == 0)
        continue; // No, keep looking.
      if((creds[i].invocation_options_required() & qopmask)
         == 0) {
       // Translate invocation_options_supported() attribute
       // into a QOP.
       qop = getQOP(creds[i].invocation_options_supported());
       break;
      } else {
       // Translate invocation_options_required() attribute
       // into a QOP.
       qop = getQOP(creds[i].invocation_options_supported());
```

```
       break;
    } // forloop
    return qop;
  }
}
```

Delegation Directive Policy

An object of **SecurityLevel2::DelegationDirectivePolicy** interface specifies whether the credentials selected may be delegated to the target or not.

```
// IDL
interface DelegationDirectivePolicy : CORBA::Policy {
                                          //Locality Constrained
    readonly attribute DelegationDirective delegation_mode;
};

// Java
package org.omg.SecurityLevel2;
public interface DelegationDirectivePolicy
     extends org.omg.CORBA.Policy
{
   public org.omg.Security.DelegationDirective
   delegation_directive();
}
```

Default Delegation Directive Policy

ORBASEC SL2 comes with a default DelegationDirectivePolicy that always returns org.omg.Security.DelegationDirective.SecNoDelegate just to be on the safe side.

Establish Trust Policy

An object of **SecurityLevel2::EstablishTrustPolicy** interface specifies the invocation conditions on establishing client or target trust.

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

// Java
package org.omg.SecurityLevel2;
public interface EstablishTrustPolicy
     extends org.omg.CORBA.Policy
{
   public org.omg.Security.EstablishTrust trust();
}
```

If the value of the **trust_in_client** field of the **trust** attribute is true, then client must select a mechanism that supports client side authentication. If the value is false, it does not matter.

If the value of the **trust_in_target** field of the **trust** attribute is true, then the client must select a mechanism that is capable of getting the target to authenticate itself before the invocation can be made. If it is false, whether the target does authenticate itself does not matter.

Default Establish Trust Policy

ORBASEC SL2 comes with a default Establish Trust policy that is set on the initial thread of execution and is inherited from every descendant thread until it is explicitly set. The default **EstablishTrustPolicy** that is set dynamically sets the **EstablishTrust** to match the invocation options that are required or supported by the credentials on the thread based **InvocationCredentialsPolicy**.

This policy serves as an attempt to use the current credentials that have been created by the application, without having the application writer to have to think about policy objects.

The semantics of this “dynamic” Establish Trust Policy roughly follows the implementation below:

```
// Java
package orbasec.seclev2;
import org.omg.Security.*;
import org.omg.SecurityLeve2.*;

public class DynEstablishTrustPolicy
  extends    orbasec.corba.LocalObject,
  implements org.omg.SecurityLevel2.EstablishTrustPolicy
{
        .....


  private EstablishTrust
  getEstablishTrust(short association_options)
  {
    // definition of function that translates the association
    // options to an EstablishTrust structure

  }
```

```
  // Policy Function
  public EstablishTrust
  trust()
  {
    // Get the invocation credentials policy
    InvocationCredentialsPolicy invocp =
        InvocationCredentialsPolicyHelper.narrow(
             current.get_policy(
                  SecInvocationCredentialsPolicy.value);
    // Create an array of strings of each credentials
    // mechanism.
    Credentials[] creds = invocp.creds();
    EstablishTrust trust = new EstablishTrust(true,true);
    int etmask = EstablishTrustInClient.value |
                 EstablishTrustInTarget.value;
    for(int i =0; i < creds.length; i++) {
      // Can we even use the credentials?
      if(creds[i].invocation_options_supported() == 0)
        continue; // No, keep looking.
      if((creds[i].invocation_options_required() &etpmask)
         == 0) {
       // Translate invocation_options_supported() attribute
       // into an EstablishTrust.
       trust = getEstablishTrust(
                   creds[i].invocation_options_supported());
       break;
      } else {
       // Translate invocation_options_required() attribute
       // into a EstablishTrust.
       trust = getEstablishTrust(
                   creds[i].invocation_options_required());
       break;
    } // forloop
    return trust;
  }
}
```

Invocation Policy Analysis

On every first invocation of a operation on an object the ORB sets up a secure association with a target via its object reference. The properties of the secure association depend upon two things. Firstly, it depends upon the policies that are placed on the object references using the object's pseudo operation, **set_policy_overrides** (**_set_policy_overrides** in the Java Mapping) Secondly, it depends upon the policies that are set as the thread's default policies by adding them using the **Current** object's **set_overrides** operation. The security services does an analysis of those policies to select a mechanism, quality of protection, trust establishment, delegation directive, and invocation credentials that are compatible with the security components of the target's IOR.

In ORBASEC SL2 the **Current** object holds a default policy for each of the Mechanism Policy, Invocation Credentials Policy, QOP Policy, Delegation Directive Policy, and Establish Trust Policy. If any of the five aforementioned policies does not exist on the particular object reference, it is taken from the **Current** object's **get_overrides** operation. Therefore, a value for each of the attributes listed in the policies will always have a value when a secure invocation needs to be established.

The following decision procedure is used in finding a mechanism, a compatible **Credentials** object, and a security component from the targets IOR from the policies. This decision procedure is part of the CORBA Security Specification and is repeated here for your benefit.

```
For each mechanism type in the MechanismPolicy {
    Select a matching security component in the targets IOR by the mechanism
    type.
    If a matching component is found {
    Find a credentials object in the credentials list that supports the
            mechanism.
    If a credentials object is found and it supports
            the QOP Policy,
            the Delegation Directive Policy,
            and the EstablishTrust Policy {
                If the association options implied by all policies are supported
                by the selected security component in the IOR and all the
                required association options of security component are satisfied {
                    Use the selected attributes to set up the secure association.
                } else {
                    Find another credentials object and continue.
```

```
            }
        } else {
            Find another credentials object and continue.
        }
    } else {
        Get the next mechanism type from the MechanismPolicy and continue.
    }
    If no mechanism can be found {
        Raise a CORBA:NO_PERMISSION with an informative error message.
    }
}
```

Specific Policies on Object References

Setting the specific policies to use on an object reference is done in Java by using the **_set_policy_overrides** method on the object reference. A Java example follows:

```
// Java
org.omg.CORBA.Object a_object = // Some target object

org.omg.SecurityLevel2.MechanismsPolicy mechpol =
                                // a mechanism policy
org.omg.SecurityLevel2.DelegationDirectivePolicy delpol =
                                // a delegation directive policy
org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[2];

policies[0] = mechpol;
policies[1] = delpol;

org.omg.CORBA.Object b_object =
                        a_object._set_policy_overrides(
                            policies,
                            org.omg.CORBA.ADD_OVERRIDE.value);
```

The **b_object** variable contains a completely new object reference to the same object to which the **a_object** refers. However, their invocation policies may be different. Depending on the policies applied to the **b_object** reference, invocations made with the **a_object** reference and the **b_object** reference can have completely different security association attributes.

Setting Default Policies

Default policies are policies are not set specifically on the object reference. ORBASEC SL2 gets the default policies off of the **Current** object's **get_overrides** operation. An application programmer sets the default policies by setting them on the **Current** object by using its **set_overrides** operation.

Since the policy override mechanism has not yet been standardized for Current at this time, (it is awaiting agreement between the POA and the Messaging groups), setting the default policies is an ORBASEC SL2 extension; and therefore the **get_overrides** and **set_overrides** operations are found on the ORBASEC SL2 **SecLev2::Current** interface. See "ORBAsec SL2 Extentions to Current" on page 60.

```
// Java
org.omg.SecurityLevel2.MechanismsPolicy mechpol =
                                 // a mechanism policy
org.omg.SecurityLevel2.DelegationDirectivePolicy delpol =
                                // a delegation directive policy
org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[2];

policies[0] = mechpol;
policies[1] = delpol;

orbasec.SecLev2.Current current =
                               // Get SecurityCurrent Object
current.set_overrides(policies,
                        org.omg.CORBA.ADD_OVERRIDE.value);
```

ORBAsec SL2 Specific Policies

ORBAsec SL2 has the following Policies above and beyond standard SecurityLevel2 policies:

TrustedAuthorityPolicy

The Trusted Authority Policy limits the SL2 verification of authentication to specific authorities. The CORBA credentials model allows you to see the immediate principal, via the **AccessId** security attribute. However, seeing further than that,

such as an X509Certificate chain, takes special interfaces, see “Extensions for ORBAsec SL2-SSL Credentials” on page 123 for details.

You can use a **TrustedAuthorityPolicy** to have the system automatically accept authentication from authorities that you trust. The interface for the **TrustedAuthorityPolicy** is as follows:

```
#pragma prefix “orbasec”

module SecLev2
{
  struct TrustedAuthority {
     Security::MechanismType    mechanism;
     Security::Opaque           security_name;
     long                       auth_distance;
  };

  typedef sequence<TrustedAuthority> TrustedAuthorityList;

  struct TrustedAuthorityContent {
     TrustedAuthorityList    own_trusted_authorities;
     TrustedAuthorityList    client_peer_trusted_authorities;
     TrustedAuthroityList    server_peer_trusted_authorities;
  };

  interface TrustedAuthorityPolicy : CORBA::Policy {
     readonly attribute TrustedAuthorityList
                             own_trusted_authorities;
     readonly attribute TrustedAuthorityList
                             client_peer_trusted_authorities;
     readonly attribute TrustedAuthorityList
                             server_peer_trusted_authorities;
   };
}
```

TrustedAuthority

This structure holds a description of a trusted authority. It has the **mechanism** name, such as “Kerberos” or “SSL” that the Opaque encoded (see “Opaque Encodings” on page 185) **security_name** is taken as a trusted authority. The **auth_distance** field is a distance in a metric specific to the mechanism of the maximum allowable distance between the principal and the authority.

For SSL, the **mechanism** field must be “SSL”. The **security_name** field must be the Opaque encoding of the authority’s DER encoded Directory Name (i.e. **Opaque.DirectoryName**). The **auth_distance** field carries the number of certificates in a certificate chain. A distance of zero means that the distance between the principal and the trusted authority is zero. Therefore, the trusted authority must be the principal, and since the issuer’s certificate must be in the chain as well, its certificate must be self-signed.

For Kerberos, the **mechanism** field must be “Kerberos”. The **security_name** must be the Opaque encoding of a Kerberos name (i.e. **Opaque.KerberosName**). The name must be in the form of the principal of the Ticket Granting Ticket service for a Kerberos Realm, (e.g. “krbtgt/MYREALM.COM@MYREALM.COM”). The **auth_distance** not defined, and it is ignored (for now).

TrustedAuthorityContent

This structure is used to create a Trusted Authority Policy. The structure contains three lists of trusted authorities.

The first list, **own_trusted_authorities**, is used for verification of “own” credentials using the **PrincipalAuthenticator** object.

The second list, **client_peer_trusted_authorities**, lists the authorities that are trusted on the server side, should the capsule be a client during an invocation.

The third list, **server_peer_trusted_authorities**, lists the authorities that are trusted on the client side, should be capsule be servicing a remote request on one of its objects.

A class that implements the **TrustedAuthorityPolicy** interface is in the **orbasec.corba** package. The constructor for the class takes the **TrustedAuthorityPolicyContent** structure. The interface of this class in Java and its constructor is the following:

```
// Java
package orbasec.corba;
public class TrustedAuthorityPolicy
    implements LocalObject,
               orbasec.SecLev2.TrustedAuthorityPolicy
{
   // Constructor
   public TrustedAuthorityPolicy(
       orbasec.SecLev2.TrustedAuthorityPolicyContent policy
   );

   public orbasec.SecLev2.TrustedAuthority[]
   own_trusted_authorities();

   public orbasec.SecLev2.TrustedAuthority[]
   client_peer_trusted_authorities();

   public orbasec.SecLev2.TrustedAuthority[]
   server_peer_trusted_authorities();
}
```

In the Absence of a Trusted Authority Policy

If a trusted authority policy has a trusted authority list of length zero for a particular authentication type, own, client peer, or server peer, all authorities are considered "trusted" for that particular type of authentication. In the absence of a trusted authority policy, all authorities are trusted for all the authentication types.

For Kerberos, the absence of a trusted authority policy for a particular authentication type means that all principals that successfully authenticate are accepted.

For SSL, the absence of a trusted authority policy for a particular authentication type means that all principals whose certificate chains verify are accepted. However, verification in this case requires that X509 certificates from the principal up to and including a *root* certificate, which is a self-signed certificate, must be present in the principal's certificate chain.

CHAPTER 8

Security Replaceable

Security Replaceable

This section outlines the Security Replaceable module components, which are able to be replaced within the SECIOP protocol. See "Adding your own Security Mechanisms" on page 55 for more details about how to add your own SecurityReplaceable module to ORBASEC SL2.

If the interfaces in this document are adhered to and the semantics of the operations and attributes specified are strictly followed, an interested party may build their own Security Replaceable Module Component and add them into the SECIOP protocol by the new module's vault into ORBASEC SL2.

We use the term "Vault" to refer to the Security Replaceable components, **Vault**, **SecurityContext**, and **SecurityLevel2::Credentials**, since all of these components must be heavily integrated behind the interfaces with each other. The **Vault** creates objects that adhere to the **SecurityLevel2::Credentials** interface, and the **SecurityContext** interface.

The **Vault** and **SecurityContext** are used by the ORBASEC SL2 SECIOP machinery, but only the **Credentials** is exposed to the application programmer. Therefore, care must be taken by the implementer of a **SecurityLevel2::Credentials** object to ward off user mistakes and recognize bad arguments to parameters or attribute set-

tings. The following diagram, Figure 3 on page 144, illustrates the use relationships between the application visible components, the Security Replaceable components, and the ORBASEC SL2 internal components. The components with the thicker lines are Security Replaceable Components.

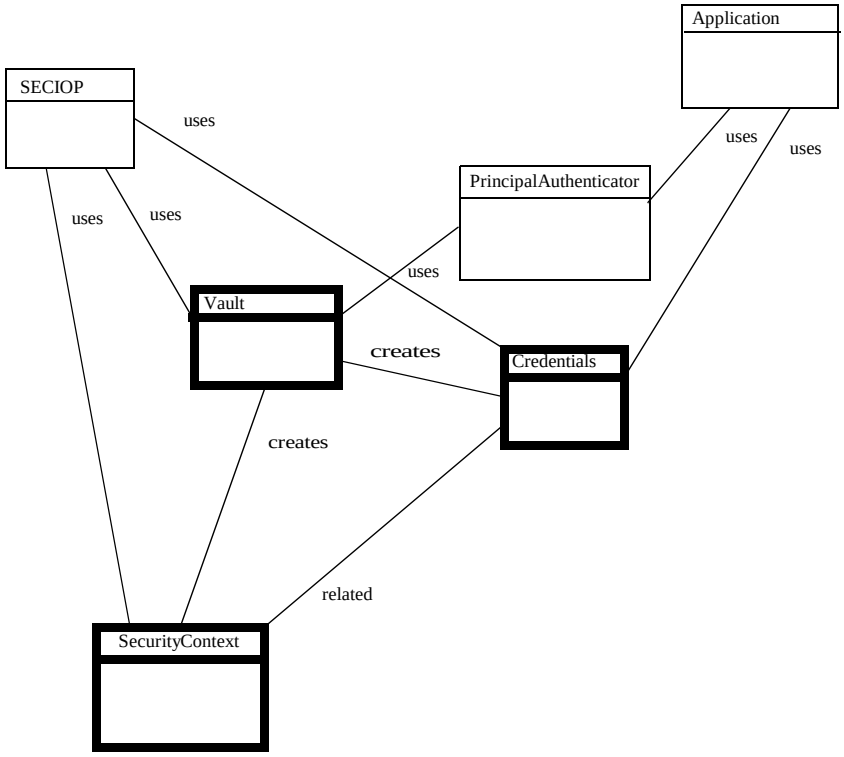


FIGURE 3. Security Replaceable Components

The Vault

The **Vault** is the object that creates **Credentials** and **SecurityContext** objects. The **Vault** creates **Credentials** on behalf of the **PrincipalAuthenticator** object that is a

default component of the SL2 machinery. The **Vault** is also called upon to create a **ServerSecurityContext** accepting secure association to targets and from clients, and **ClientSecurityContext** objects for clients initiating secure association to targets.

The **Vault**'s operations are described below:

init_security_context

This operation is used by the ORBASEC SL2 SECIOP machinery when a new secure association is needed to communicate with a client. Its outputs are required to be a GSS compliant Initial token and a **ClientSecurityContext** object.

```
// IDL
Security::AssociationStatus init_security_context(
    in  SecurityLevel2::Credentials     invoc_creds,
    in  Security::Opaque                target_security_name,
    in  Object                          target,
    in  Security::DelegationMode        delegation_mode,
    in  Security::OptionsDirectionPairList
                                        association_options,
    in  Security::MechanismType         mechanism,
    in  Security::Qpaque                mech_data,
    in  Security::ChannelBindings       chan_bindings,
    out Security::OpaqueBuffer          security_token,
    out ClientSecurityContext           security_context
);

// Java
public org.omg.Security.AssociationStatus
init_security_context(
  org.omg.SecurityLevel2.Credentials  invoc_creds,
  byte[]                              target_security_name,
  org.omg.CORBA.Object                target,
  org.omg.Security.DelegationMode     delegation_mode,
  org.omg.Security.OptionsDirectionPair[]
                                      assocation_options,
  String                              mechanism,
  byte[]                              mech_data,
  org.omg.Security.ChannelBindings    chan_bindings,
  org.omg.Security.OpaqueBufferHolder security_token,
  ClientSecurityContextHolder         security_context
);
```

There are a range of inputs. Not all of the parameters listed for this operation are used by ORBASEC SL2, i.e. given meaningful values.

creds

This parameter is given the **Credentials** object with which to create the security context. This parameter may be a **ReceivedCredentials** from a **SecurityContext** object, or it may be an "own" **Credentials** object created by this **Vault**. It cannot be a **TargetCredentials** object. The ORBAsec SL2 only makes sure that this **Credentials** object is compatible with this **Vault**, using the mechanism attribute of the **Credentials** object.

target_security_name

This parameter is the name of the target that will be used to set up the association. This name is pulled from the selected security component from the IOR of a target. This name is not uniquely specific to any one target object, as one target name may service many objects. The constraints on the value of this argument that ORBASEC SECIOP machinery will adhere with respect to the argument given to this parameter are:

- The **target_security_name** is the security name of the target according to **mechanism** selected.
- The **target_security_name** will be the same security name found in the **mech_data** argument, as the **mech_data** argument is the selected security component from the IOR.

target

This parameter is not used by ORBASEC SL2 as the internal architecture does not yield the target object reference used to make the invocation at the transport level. Also, security associations established with a principal that is represented by a security name, which is not guaranteed to reference a single target object. ORBASEC SL2 may choose, based on policy analysis at the time of an invocation, to reuse a security context.

delegation_mode

This argument specifies a capability of no delegation, simple delegation, or composite delegation that will be used. It is guaranteed by ORBASEC SL2 policy analysis that the values presented to this parameter have the values of the delegation

mode the credentials being used will support, such as from the **invocation_options_required** attribute on **Credentials**.

association_options

ORBASEC SL2 gives this parameter an argument that is a sequence that contains only one **OptionsDirectionPair** structure. ORBASEC SL2 gives an argument to this parameter adhering to the following constraints:

- The **communications_direction** attribute of that structure will be **SecCommunicationsDirectionBoth**.
- The value of the **association_options** attribute will be suitably selected, such that it will adhere correctly to the **mech_data** containing the **target_supports** and **target_requires** attributes.
- The value of the **association_options** attribute will correctly adhere to the options that are supported from **invocation_options_supported** attribute and **invocation_options_required** attribute of the **Credentials** object specified in the **invoc_creds** parameter.

The ORBASEC SL2 policy analysis will guarantee that the value of the association options will fit in with the capabilities of the target and the credentials being used to set up the security context.

mechanism

This parameter is the selected mechanism to use to set up the secure association. ORBASEC SL2 gives an argument to this parameter adhering to the following constraints:

- The **mechanism** is the value of the mechanism name constructed from the security component in the IOR that was selected. The selected security component is in the **mech_data** argument.

mech_data

ORBASEC SL2 gives as a value to this argument the security component of the selected **mechanism** from the IOR.

chan_bindings

This argument is used by the SECIOP machinery, which runs over TCP/IP. The Channel Bindings that are supported are those of the GSS_C_AF_INET address type, which stipulate the network byte order host IP addresses of the client and the server.

security_token

This parameter is an output parameter. Any implementation is required to make this token a GSS compliant Initial Token, [4, Section 15.9] to guarantee interoperability. However, if one chooses to build and install a proprietary **Vault** for all communicating ORBs in its enterprise, then this token just needs to adhere to a format compatible with the **in_token** of the **accept_security_context** operation of the **Vault**.

security_context

This parameter is an output parameter. The **Vault** must create a **ClientSecurity-Context** object to represent the initialized security context.

return value

Valid return values for this operation are **Security::SecAssocSuccess** if the **Vault** was successful in creating a security token and a **ClientSecurityContext** in an initialized state. It must not return **Security::SecAssocContinue**. It may return **Security::SecAssocFailure** should it fail to create a token and a **ClientSecurityContext** for some reason. However, we would prefer that a CORBA system exception be raised with an informative message detailing the error encountered.

accept_security_context

This operation is called upon by the ORBASEC SL2 SECIOP machinery when a SECIOP **EstablishContext** message is received.

```
// IDL
Security::AssociationStatus accept_security_context(
    in  SecurityLevel2::CredentialsList  creds_list,
    in  Security::ChannelBindings        chan_bindings,
    in  Security::OpaqueBuffer           in_token,
    out Security::OpaqueBuffer           out_token,
    out ServerSecurityContext            security_context
  );

// Java
org.omg.Security.AssociationStatus
accept_security_context(
  org.omg.SecurityLevel2.Credentials[]      creds_list,
  org.omg.Security.ChannelBindings          chan_bindings,
  org.omg.CORBA.Security.OpaqueBuffer       in_token,
  org.omg.CORBA.Security.OpaqueBufferHolder out_token,
  ServerSecurityContext                     security_context
);
```

The arguments given to this operation are as follows:

creds_list

This parameter holds the credentials that may be needed to accept the request to establish a secure association. ORBASEC SL2 gives this parameter the list of "own" credentials that were created by all the **Vault** OBJECTS. According to a discriminator on the front of the GSS Initial Token [4, Section 15.10.7] the **Vault** should be able to discern the mechanism used. The **Vault** must have the capability to search through the list of "own" credentials and find the proper ones to support the secure association.

chan_bindings

This argument is used by the SECIOP machinery, which runs over TCP/IP. The Channel Bindings that are supported are those of the GSS_C_AF_INET address type, which stipulate the network byte order host IP addresses of the client and the server.

in_token

This parameter is given the token verbatim that is extracted from the SECIOP **EstablishContext** message that is received from the client.

out_token

This token must be a buffer containing a sequence of bytes that is of the format of a GSS compliant **ContinueEstablish** or a **TargetResult** token.

security_context

This parameter must contain a newly created **ServerSecurityContext** in the appropriate state as a result of processing the **in_token**.

return value

This operation should return **Security::SecAssocSuccess** if processing the initial token results in an established secure association with the client. A SECIOP **CompleteEstablishment** message will be sent back to the client with the value of the **out_token** parameter. The **out_token** should contain a GSS compliant **TargetResult** token.

This operation should return **Security::SecAssocContinue** if processing the initial token results in creating a **ServerSecurityContext** that is not quite established (i.e. the target is requesting more authentication). A SECIOP **ContinueEstablishment** message will be sent back to the client with the value of the **out_token** parameter. The **out_token** parameter should contain a GSS compliant **ContinueEstablish** token.

This operation may return **Security::SecAssocFailure** if processing the initial token yields an error. However, we prefer that a CORBA system exception be raised with an informative message as to the error encountered. In either case a SECIOP **DiscardContext** message will be sent back to the client.

acquire_credentials

This operation is used by the **PrincipalAuthenticator** to create “own” credentials. In ORBASEC SL2 the **PrincipalAuthenticator::authenticate** operation makes the call to the **Vault::acquire_credentials** operation almost as a pass through operation. The **PrincipalAuthenticator** acts as the application’s delegate to the **Vault**, but places the created “own” credentials on the **Current** object’s own credentials list.

Note. The current **PrincipalAuthenticator** in ORBASEC SL2 does not to any parameter integrity checking.

The **acquire_credentials** operation's interface is described below.

```
// IDL
Security::AuthenticationStatus acquire_credentials (
    in  Security::AuthenticationMethod    method,
    in  Security::MechanismType           mechanism,
    in  Security::Opaque                  security_name,
    in  Security::Opaque                  auth_data,
    in  Security::AttributeList           privileges,
    out SecurityLevel2::Credentials       creds,
    out Security::Opaque                  continuation_data,
    out Security::Opaque                  auth_specific_data
);

// Java
org.omg.Security.AuthenticationStatus
acquire_credentials(
    int                                   method,
    String                                mechanism,
    byte[]                                security_name,
    byte[]                                auth_data,
    org.omg.Security.SecAttribute[]       privileges,
    org.omg.SecurityLevel2.CredentialsHolder
                                          creds,
    org.omg.Security.OpaqueHolder         continuation_data,
    org.omg.Security.OpaqueHolder         auth_specific_data
);
```

method

This parameter specifies method with which to authenticate the principal. The methods that are allowed in this call are specific to the implementation of the **Vault**.

mechanism

This parameter specifies mechanism with which to authenticate the principal using the **security_name** and create the credentials. The mechanisms that are allowed in this call are the mechanisms that must be supported by **Vault**.

security_name

This parameter is a byte array stating the recognized name of the principal for which to acquire credentials.

auth_data

This parameter specifies the extra data needed to authenticate the principal using the **security_name**. The format of this must be specified by the implementer of the **Vault**.

privileges

This parameter states the “extra” privileges that the application programmer wants to be authenticated along with the principal to create the credentials with those privileges authorized. Such privileges can be requesting or stating that the principal is the member of a group, or has the authorization for a particular role.

creds

This parameter is an output parameter returning the newly created “own” **Credentials** object. The **PrincipalAuthenticator** works in concert with the **Current** object and places the new credentials in the current’s own credentials list repository. These may not be fully enabled credentials as the authentication mechanism may have created interim credentials to be further passed to the **continue_credentials_acquistion** operation. The PrincipalAuthenticator will not place these **Credentials** on the “own” credentials list until a value of **SecAuthSuccess** has been returned from **acquire_credentials** or **continue_credentials_acquision**.

continuation_data

This parameter is an output parameter returning data needed to continue the authentication of the principal using the **security_name**. This may hold such data labeling a continuation context. Its output will be given to the **continue_credentials_acquistion** operation.

auth_specific_data

This parameter is an output parameter returning data that may need to be exposed to the application programmer, such as a message about what is needed to continue

the authentication. The implementer of the **Vault** will need to specify what the format is and how the application implementer may use it.

return value

The return value is one of the value of the **Security::AuthenticationStatus** enumeration type, and states whether authentication succeeded, failed, needs to be continued, or if continued, the further continuation has expired.

This operation must return a value of **Security::SecAuthSuccess** if the operation was successful and the output credentials are valid "own" credentials. It must return a value of **Security::SecAuthContinue** if the acquisition process needs to be continued. This operation should return **Security::SecAuthFailure** should the acquisition fail. However, we would prefer to the operation to raise a CORBA system exception with an informative message as to the error encountered. This operation, being the initial acquisition, must not return **Security::SecAuthExpired**.

continue_credentials_acquisition

This operation is meant to continue acquisition steps started by **acquire_credentials**, and possibly still continued by subsequent calls to **continue_credentials_acquistion**. Its interface is defined below:

```
// IDL
Security::AuthenticationStatus
                        continue_credentials_acquisition(
    in     Security::Opaque             response_data,
    in     SecurityLevel2::Credentials  creds,
    out    Security::Opaque             continuation_data,
    out    Security::Opaque             auth_specfic_data
);

// Java
public org.omg.Security.AutenticationStatus
continue_credentials_aquision(
  byte[]                                response_data,
  org.omg.SecurityLevel2.Credentials    creds,
  org.omg.Security.OpaqueHolder         continuation_data,
  org.omg.Security.OpaqueHolder         auth_specific_data
);
```

response_data

The argument given to this parameter is data in the format specified by the implementer of the **Vault** that pertains to the mechanism of credentials being used to continue the acquisition of the credentials.

creds

The argument given to this parameter will be credentials returned from **acquire_credentials** or subsequent calls to **continue_credentials_acquisition**. If the operation returns a value of **Security::SecAuthSuccess**, the credentials will be fully enabled and placed on **Current**'s own credentials list by the **PrincipalAuthenticator**.

continuation_data

If the operation returns **Security::SecAuthContinue**, this output value should be used in the subsequent call to **continue_credentials_acquisition**.

auth_specific_data

If the operation returns **Security::SecAuthContinue**, this output value should be used in the subsequent call to **continue_credentials_acquisition**.

return value

This operation must return a value of **Security::SecAuthSuccess** if valid "own" credentials are created. The **PrincipalAuthenticator** will place these credentials in the **Current** object's own credentials list.

This operation must return a value of **Secuirty::SecAuthContinue** if subsequent calls to **continue_credentials_acquistion** are still needed.

This operation must return a value of **Security::SecAuthExpired** if the continuation has gone on too long and for some reason can no longer be continued.

This operation must return a value of **Security::SecAuthFailure** if the credentials cannot be created. However, we prefer that a CORBA system exception be raised with an informative message as to the error encountered.

get_supported_mechs

This operation should return the mechanisms and supported options for which the **Vault** is capable of creating credentials and security contexts.

```
// IDL
Security::MechandOptionsList get_supported_mechs();

// Java
org.omg.Security.MechandOptions[]
get_supported_mechs();
```

get_supported_authen_methods

This operation should return the authentication method tags that are supported by this vault for a particular mechanism that this **Vault** supports. If the Vault has advertised that it supports a mechanism type, from its **get_supported_mechs** operation, this call must return a list of valid tags for the mechanism that it supports for the call to **acquire_credentials**. We suggest that the tag value of zero be used to mean “default”.

```
// IDL
Security::AuthenticationMethod get_supported_authen_methods(
    in Security::MechanismType       mechanism
);

// Java
pubilc int[]
get_supported_authen_methods(String mechanism);
```

supported_mech_oids

This operation should return the ISO standard OIDs for the supported mechanisms for which the **Vault** is capable of creating credentials and security contexts. An OID of a specific mechanism is always contained in the header of a GSS Initial Token which is given to the **accept_security_context** operation. The OIDs are advertised here by the **Vault** so that the SECIOP machinery can determine the whether the **Vault** can handle a specific GSS Initial Token, or direct it to a **Vault** that can.

```
// IDL
Security::OIDList supported_mech_oids();
```

```
// Java
byte[][]
supported_mech_oids();
```

Credentials

The **SecurityLevel2::Credentials** interface is the base type for own credentials and received credentials. The “own” type credentials is the **SecurityLevel2::Credentials** interface itself, while **SecurityLevel2::ReceivedCredentials** and **SecurityLevel2::TargetCredentials** extends it.

A **Credentials** object holds information pertaining to the authenticated identity of the subject of the credentials, i.e. the principal, via the security name, by either **acquire_credentials** or **accept_security_context** operation of the **Vault**.

```
// IDL
interface Credentials { // Locality Constrained
  Credentials copy();

  void destroy():

  readonly attribute Security::CredentialsType
                                  credentials_type;

  readonly attribute Security::AuthenticationState
                                  authentication_state;

  readonly attribute Security::MechanismType  mechanism;

  attribute Security::AssociationOptions
                                  accepting_options_supported;
  attribute Security::AssociationOptions
                                  accepting_options_required;
  attribute Security::AssociationOptions
                                  invocation_options_supported;
  attribute Security::AssociationOptions
                                  invocation_options_required;
  boolean get_security_feature(
    in   Security::CommunicationDirection   direction,
    in   Security::SecurityFeature          feature
  );
```

```
  boolean set_attributes (
    in  Security::AttributeList       requested_attributes,
    out Security::AttributeList       actual_attributes
  );

  Security::AttributeList get_attributes(
    in  Security::AttributeTypeList   attributes
  );

  boolean is_valid (
    out Security::UtcT                expiry_time
  );

  boolean refresh(
    in  Security::Opaque              refresh_data
  );
};
```

The attributes and operations of the **Credentials** interface are:

copy

This operation may be called on by the user to copy credentials. The credentials may be modified by the user, so care should be taken to create a new **Credentials** object preserving information in any context it may be place in for which a copy of the **Credentials** is deemed warranted.

ORBASEC SL2 SECIOP machinery makes no calls to the **copy** operation of **Credentials**.

The implementer should take care to make copies of credentials in the various places they are produced and housed in the context of the replaceable module. For example, the **Credentials** stored in the **client_credentials** attribute on the **ClientSecurityContext** should be a copy of the **Credentials** used to create the context, which are may be one of the user accessible "own" **Credentials** on the **Current** object. The application may change the option attributes of user accessible **Credentials** object and then alter the credentials hanging off the **ClientSecurityContext** object.

The implementer should detail how the general copies of **Credentials** objects are affected by the **destroy** operation on one of the copies.

```
// IDL
Credentials copy();

// Java
public org.omg.SecurityLevel2.Credentials copy();
```

destroy

This operation is called upon destroy the credentials object so that applications can do their own credentials management. This also gives the **Credentials** operation the ability to do some memory management and take care of loose ends.

ORBASEC SL2 SECIOP machinery makes no calls on the **destroy** operation.

```
// IDL
void destroy();

// Java
public void destroy();
```

credentials_type

This attribute contains the value discerning whether the credentials are of the "own" or "received" type.

```
// IDL
readonly attribute Security::CredentialsType
                                                credentials_type;
// Java
public org.omg.Security.CredentialsType
credentials_type();
```

This operation must return **Security::SecCredentialsType::SecOwnCredentials** if the **Credentials** is of the "own" credentials type, **Security::SecCredentialsType::SecReceivedCredentials** if the **Credentials** object is of the "received" credentials type and can be narrowed to a **ReceivedCredentials** object, and **Security::SecCredentialsType::SecTargetCredentials** if the **Credentials** object is of the "target" credentials type and can be narrowed to a **TargetCredentials** object.

authentication_state

Since **Credentials** objects may take several operations to fully become initialized this read-only attribute serves as an indication of the authentication state, which is the same as the result returned from **PrincipalAuthenticator::authenticate** and **PrincipalAuthenticator::continue_authentication** operations.

```
// IDL
readonly attribute Security::AuthenticationStatus
                                              authentication_state;
// Java
public org.omg.Security.Authenticationstatus
authentication_state();
```

This attribute must have the value of **Security::AuthenticationStatus::SecAuthSuccess** if the **Credentials** are fully initialized. It must have the value of **Security::AuthenticationStatus::SecAuthContinue** if subsequent calls to **PrincipalAuthenticator::continue_authentication** are needed. It must have the value **Security::AuthenticationStatus::SecAuthFailure** if the continuing authentication of the **Credentials** has failed. It must have the value of **Security::AuthenticationStatus::SecAuthExpired** if the continuing authentication of the **Credentials** is no longer viable.

mechanism

This read only attribute specifies the symbolic name security mechanism and the symbolic name of the cipher suites that the credentials support.

```
// IDL
readonly attribute Security::MechanismType mechanism;

// Java
public String mechanism();
```

Please see the section on “Mechanism” on page 89. for detail. Also, please see the JavaDoc built documentation on **orbasec.corba.MechUtil** to see how you may register symbolic names for your mechanisms and ciphers into that facility.

accepting_options_supported

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the server side. It also serves as the value that is

placed in the "**target_supports**" field of the security component (should one exist) for the particular security mechanism in an objects's IOR.

Setting of the attribute's value to an illegal set of **Security::AssociationOptions** must raise a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    accepting_options_supported;
// Java
public short accepting_options_supported();
public void accepting_options_supported(short opts);
```

Accepting options supported must be non-zero to be used with **SecLev2::Current::set_accepting_credentials** operation. The absolute minimum in security terms that any credentials object can have in supported options to establish an association is:

NoProtection + NoDelegation

For most security mechanisms, "received" credentials object must have accepting options of zero. This attribute having a value of zero simply states that this credentials object cannot be used to establish secure associations on the server side.

After Credentials are fully initialized the user can change the options these credentials support. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the server side. They cannot be set to less than the **accepting_options_required** attribute. If one must decrement the options that are supported, one must set the required options first. The options that are supported cannot be set to more than the options that the credentials were created with. Credentials are created with their maximum supported options set in this attribute. The implementer should take care to enforce these rules.

accepting_options_required

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the server side. It also serves as the value that is placed in the "**target_requires**" field of the security component (should one exist) for the particular security mechanism in an objects's IOR.

Setting of the attribute's value to an illegal set of **Security::AssociationOptions** must raise a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    accepting_options_required;
// Java
public short accepting_options_required();
public void accepting_options_required(short opts);
```

Accepting options required may be zero.

After Credentials are fully initialized the user can change the options these credentials require. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the server side. They cannot be set to more than the **accepting_options_supported** attribute. If one must augment the options that are required, one must set the supported options first. The implementer should take care to enforce these rules.

invocation_options_supported

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the client side.

Setting of the attribute's value to an illegal set of **Security::AssociationOptions** must raise a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                  invocation_options_supported;
// Java
public short invocation_options_supported();
public void invocation_options_supported(short opts);
```

Invocation options supported must be non-zero to be used with an **SecurityLevel2::InvocationCredentialsPolicy**. The absolute minimum in security terms that any credentials object can have in supported options to establish an association is:

NoProtection + NoDelegation

In the case of delegation, "received" credentials may have supported invocation options. This attribute having a value of zero simply states that this credentials object cannot be used to establish secure associations on the client side.

After Credentials are fully initialized the user can change the options these credentials support. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the client side. They cannot be set to less than the **invocation_options_required** attribute. If one must decrement the options that are supported, one must set the required options first. The options that are supported cannot be set to more than the options that the credentials were created with. Credentials are created with there maximum supported options set in this attribute. The implementer should take care to enforce these rules.

invocation_options_required

This attribute gives control over certain capabilities of the credentials object when setting up secure associations on the client side.

Setting of the attribute's value to an illegal set of **Security::AssociationOptions** must raise a **CORBA::BAD_PARAM** exception.

```
// IDL
attribute Security::AssociationOptions
                                    invocation_options_required;
// Java
public short invocation_options_required();
public void invocation_options_required(short opts);
```

Invocation options required may be zero.

After Credentials are fully initialized the user can change the options these credentials require. Changing the options alters the characteristics of the credentials when they are used to establish secure associations on the client side. They cannot be set to more than the **invocation_options_supported** attribute. If one must augment options that are required, one must set the supported options first. The implementer should take care to enforce these rules.

get_security_feature

This operation returns a boolean that represent a security feature's state of the credentials. It is not used by any ORBAsec SL2 SECIOP machinery. It is a user level interface.

```
// IDL
boolean get_security_feature(
    in Security::CommunicationDirection   direction,
    in Security::SecurityFeature          feature
);

// Java
public boolean get_security_features(
    int                                direction,
    org.omg.Security.SecurityFeature feature
);
```

If the communication direction is **Security::CommunicationDirection::SecDirectionRequest**, the feature returned should be for invocation, i.e. as a client. If the communication direction is **Security::CommunicationDirection::SecDirectionReply**, the feature return should be for the accepting requests, i.e. as a server.

We suggest that the values returned for a given feature mirror the option state in the **invocation_options_supported** attribute in the **SecDirectionRequest** case, and the **accepting_options_supported** attribute in the **SecDirectionReply** case. However, your mechanism may specify otherwise.

set_attrbiutes

This operation is intended for use in attribute management of the particular credentials. Its meaning is defined to diminish the attributes of the credentials in the context of the mechanism's ability. The implementer should take care to notice that the requested attributes is a subset of the exact attributes that would be returned from the **get_attributes** operation.

The **set_attributes** operation's interface is below:

```
// IDL
boolean set_attributes(
  in   Security::AttributeList   requested_attributes,
  out  Security::AttributeList   actual_attributes
);

// Java
public boolean set_attributes(
    org.omg.Security.SecAttribute[]      requested_attributes,
    org.omg.Security.AttributeListHolder actual_attributes
);
```

Users may call this operation if they want to subsequently remove security attributes from the Credentials. The implementer should take care to make sure that the value given to the requested_attributes is a subset of the exact attributes that would be returned from the **get_attributes** operation. It is realized that some attributes that are not supplied may not be able to be removed from the credentials. Yet, the operation may be successful enough not to warrant the raising of an exception. This operation should return true if the operation is successful and all the attributes of the credentials now match the requested attributes. This operation should return false if the operation is successful, but some of the requested attributes did not include attributes that cannot be removed. The **actual_attributes** parameter always returns all the attributes of the credentials. If the operation is not successful a system exception of **CORBA::BAD_PARAM** should be raised.

get_attributes

This operation returns an unordered sequence of security attributes that belong to the credentials. Although there is a standard for the attribute types and the values to which they refer, no standardization effort is underway to define the format of the values of the particular attributes.

```
// IDL
Security::AttributeList get_attributes(
  in   Security::AttributeTypeList     attributes
);

// Java
public org.omg.Security.SecAttribute[] get_attributes(
     org.omg.Security.AttributeType[]  attributes
);
```

We strongly suggest that you use the **orbasec.corba.Opaque** class for Opaque encodings of the **defining_authority** and **value** fields of the **SecAttribute**. See “Opaque Encodings” on page 185.

is_valid

This operation returns a boolean value indicating whether the credentials are still valid. The output parameter returns the time of expiration.

```
// IDL
boolean is_valid(
  out   Security::UtcT expiry_time
);

// Java
public boolean is_valid(
    org.omg.TimeBase.UtcTHolder expiry_time
)
```

refresh

This operation is intended to renew a credentials before it may expire. It returns a boolean value indicating the success of the renewal.

```
// IDL
boolean refresh(
   in  Security::Opaque  refresh_data
);

// Java
public boolean refresh( byte[] refresh_data );
```

We suggest that if your mechanism cannot refresh either own credentials or received credentials, that this operation raise a **CORBA::BAD_OPERATION** exception.

Received Credentials

On the target side a **ReceivedCredentials** object represents a secure association between the client and target. Received credentials must have more information than “own” credentials. An object implementing this interface should be returned from the call to **accept_security_context** on the **Vault**.

The interface inherits from the **Credentials** interface, and in the case of using the received credentials for invocations, the invocation features, operations, and attributes of the **Credentials** object have the same meaning. Of course, the **credentials_type** attribute is set to **SecReceivedCredentials**. Its interface is defined below:

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

accepting_credentials

This read-only attribute is the **Credentials** objects that was used to establish the secure association with the client. It should be one of the credentials objects that was given to **accept_security_context** of the **Vault**.

```
// IDL
readonly attribute Credentials accepting_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
accepting_credentials();
```

association_options_used

This read-only attribute states the association options that were used to make the association with the **accepting_credentials**. This value should be a value somewhere between the **accepting_options_required** and the **accepting_options_supported** of the **accepting_credentials**.

```
// IDL
readonly attribute Security::AssociationOptions
                                     association_options_used;
// Java
public short accociation_options_used();
```

delegation_state

This read-only attribute is the value of the delegation state of the *client's own credentials*. It states whether the immediate invoking principal of the operation is the initiator or a delegate of some other principal.

```
// IDL
readonly attribute Security::DelegationState delegation_state;

// Java
public org.omg.Security.DelegationState
delegation_state();
```

Note – For some security mechanisms, this information is indeterminable. When this information is indeterminable, impersonation is assumed; and therefore, this attribute must have the value of **SecInitiator**.

delegation_mode

This read-only attribute states the delegation mode of the received credentials. It stipulates that the credentials are in the a delegation mode of:

- No delegation mode (**SecDelModeNoDelegation**), where they can not be used for invocations.
- Simple delegation mode (**SecDelModeSimpleDegation**), where the credentials can be indiscriminately used on the client's behalf.
- Composite delegation (**SecDelModeCompositeDelegation**) where the credentials have some sort of composite ability, such as a trace, a combination of privileges, etc.

```
// IDL
readonly attribute Security::DelegationMode delegation_mode;

// Java
public org.omg.Security.DelegationMode
delegation_mode();
```

Target Credentials

On the target side a **TargetCredentials** object represents a secure association between the client and target from the client's point of view. Target credentials must have more information than "own" credentials. An object implementing this interface should be returned from the call to **server_credentials** attribute on **ClientSecurityContext**.

The interface inherits from the **Credentials** interface, and in the case of using the received credentials for invocations, the invocation features, operations, and attributes of the **Credentials** object have the same meaning. The **credentials_type** attribute is set to **SecTargetCredentials**. Its interface is defined below:

```
interface TargetCredentials : Credentials {
                                    // Locality Constrained
  readonly attribute Credentials     initiating_credentials;
  readonly attribute Security::AssociationOptions
                                       association_options_used;
};
```

initiating_credentials

This read-only attribute is the **Credentials** objects used to establish the secure association with the server. This **Credentials** object should be the one given to the **init_security_context** operation.

```
// IDL
readonly attribute Credentials initiating_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
initiating_credentials();
```

association_options_used

This read-only attribute states the association options that were used to make the association with the **initiating_credentials**. This value should be a value somewhere between the **invocation_options_required** and the **invocation_options_supported** of the **initiating_credentials**.

```
// IDL
readonly attribute Security::AssociationOptions
                                       association_options_used;
// Java
public short accociation_options_used();
```

Security Context

The **SecurityContext** object is the base interface for the **ClientSecurityContext** object and the **TargetSecurityContext** object.

```
interface SecurityContext {
  readonly attribute Security::ContextType   context_type;
  readonly attribute Security::ContextState  context_state;
  readonly attribute Securty::MechanismType  mechanism;
  readonly attribute boolean                 supports_refresh;
  readonly attribute Security::ChannelBindings
                                             chan_binding;
  readonly attribute SecurityLeve2::Credentials
                                             peer_credentials;

  Security::AssocationStatus continue_security_context(
    in  Security::OpaqueBuffer        in_token,
    out Security::OpaqueBuffer        out_token
  );

  void protect_message(
    in  Security::OpaqueBuffer        message,
    in  Security::QOP                 qop,
    out Security::OpaqueBuffer        text_buffer
    out Security::QpaqueBuffer        out_token
  );

  void reclaim_message(
    in  Security::OpaqueBuffer        text_buffer,
    in  Security::OpaqueBuffer        token,
    out Security::QOP                 qop,
    out Security::OpaqueBuffer        message
  );

  boolean is_valid(
    out Security::UtcT                 expiry_time
  );

  void refresh_security_context(
    in  Security::Opaque               refresh_data,
    out Security::OpaqueBuffer         out_token
  )
```

```
  boolean process_refresh_token(
    in Security::OpaqueBuffer         refresh_token
  )


  void discard_security_context(
    in  Security::Opaque               refresh_data,
    out Security::OpaqueBuffer         out_token
  )


  boolean process_discard_token(
    in Security::OpaqueBuffer         refresh_token
  );
};
```

context_type

This read-only attribute contains the discriminator that determines whether this context is a **ClientSecurityContext** or a **ServerSecurityContext**.

```
// IDL
readonly attribute Security::SecurityContextType
                                                    context_type;
// Java
public org.omg.Security.SecurityContextType
context_type();
```

context_state

This read-only attribute indicates the establishment state of the security context.

```
// IDL
readonly attribute Security::SecurityContextState
                                                   context_state;
// Java
public org.omg.Security.SecurityContextState
context_state();
```

The ORBAsec SL2 SECIOP machinery pays attention to the following states during its processing of secure associations:

SecContextInitialized

A **SecurityContext** state of **SecContextInitialized** is the initial state of a security context created by the Vault.

SecContextContinued

A **SecurityContext** state of **SecContextContinued** means the security context still needs to do continuance processing. It will not be used protect messages.

SecContextClientEstablished

A **SecurityContext** state of **SecContextClientEstablished** means the security context still needs to do continuance processing, but is able to protect messages on the client side.

An example of this situation is when mutual authentication is not needed. Once the client produces the initial token, it can be ready to protect messages without some response from the target.

Note – SECIOP protocol has no provision for being able to reclaim messages without first entering the **SecContextEstablished** state.

SecContextEstablished

A **SecurityContext** state of **SecContextEstablished** means the security context is able to protect messages and reclaim messages.

SecContextEstablishExpired

A **SecurityContext** of **SecContextEstablishExpired** means that establishment processing for the security context has expired, and it can no longer be used to accept calls to continue establishment, protect messages, or reclaim messages.

SecContextExpired

A **SecurityContext** state of **SecContextExpired** means the security context has expired, and it can no longer be used to accept calls to continue establishment, protect messages, or reclaim messages.

SecContextInvalid

A **SecurityContext** of **SecContextInvalid** means that the security context is no longer usable.

supports_refresh

This read-only attribute tells the ORBASEC SL2 SECIOP machinery whether the context may be, or has the ability to be refreshed.

```
// IDL
readonly attribute boolean supports_refresh;

// Java
public boolean supports_refresh();
```

mechanism

This read-only attribute is the **mechanism** used in the creation of the **SecurityContext**. by the **Vault**. It usually depends upon the capabilities of the **Vault** and the Credentials object(s) given to **init_security_context** or **accept_security_context**.

```
// IDL
readonly attribute Security::MechanismType mechanism;

// Java
public String mechanism();
```

chan_binding

This read-only attribute is the **chan_binding** parameter used in the creation of the **SecurityContext** by the **Vault**.

```
// IDL
readonly attribute Security::ChannelBindings chan_binding;

// Java
public org.omg.Security.ChannelBindings chan_binding();
```

peer_credentials

This attribute returns the ReceivedCredentials or TargetCredentials object that represents the secure association. If the security context is a ClientSecurityContext, the

peer credentials are that of TargetCredentials. If the security context is a ServerSecurityContext, the peer credentials are that of ReceivedCredentials.

continue_security_context

This operation is called on by SECIOP to continue security contexts. The input token is either supplied by **SECIOP::ContinueEstablishment** or **SECIOP::CompleteEstablishment** messages.

```
// IDL
 Security::AssocationStatus continue_security_context(
    in  Security::OpaqueBuffer        in_token,
    out Security::OpaqueBuffer        out_token
  );

// Java
public org.omg.Security.AssociationStatus
continue_security_context(
    org.omg.Security.OpaqueBuffer         in_token,
    org.omg.Security.OpaqueBufferHolder   out_token
);
```

protect_message

This operation is used by SECIOP to send **SECIOP::MessageInContext** messages.

```
// IDL
void protect_message(
    in  Security::OpaqueBuffer        message,
    in  Security::QOP                 qop,
    out Security::OpaqueBuffer        text_buffer
    out Security::QpaqueBuffer        out_token
);

// Java
public void protect_message(
   org.omg.Security.OpaqueBuffer       message,
   org.omg.Security.QOP                qop,
   org.omg.Security.OpaqueBufferHolder text_buffer,
   org.omg.Security.OpaqueBufferHolder out_token
);
```

reclaim_message

This operation is used by SECIOP to decode **SECIOP::MessageInContext** messages.

```
// IDL
void reclaim_message(
    in  Security::OpaqueBuffer        text_buffer,
    in  Security::OpaqueBuffer        token,
    out Security::QOP                 qop,
    out Security::OpaqueBuffer        message
);

// Java
public void reclaim_message(
    org.omg.Security.OpaqueBuffer       text_buffer,
    org.omg.Security.OpaqueBuffer       token,
    org.omg.Security.QOPHolder          qop,
    org.omg.Security.OpaqueBufferHolder message
);
```

is_valid

This operation states the expiry time of the security context should it be known. ORBAsec SL2 currently does not make use of this operation.

```
// IDL
boolean is_valid(
    out Security::UtcT                expiry_time
);

// Java
public boolean is_valid(
   org.omg.TimeBase.UtcTHolder expiry_time
);
```

refresh_security_context

This operation attempts to refresh the security context. It has one input parameter and one output parameter.

```
// IDL
void refresh_security_context(
    in  Security::Opaque               refresh_data,
    out Security::OpaqueBuffer         out_token
);

// Java
public void refresh_security_context(
    org.omg.Security.Opaque             refresh_data,
    org.omg.Security.OpaqueBufferHolder out_token
)
```

refresh_data

This parameter contains the information that may be necessary to reestablish the context.

out_token

This parameter contains the information that is to be transmitted back to the remote side in a SECIOP EstablishContext message.

Note – There is a flaw in SECIOP in the way it is supposed to reestablish a context should it expire on the target side.

ORBASEC SL2 currently does not make use of this operation.

process_refresh_token

This operation attempts to process a refresh token produced by the **refresh_security_context** operation of the remote side of the security context. It has one input parameter.

```
// IDL
boolean process_refresh_token(
    in Security::OpaqueBuffer         refresh_token
);

// Java
public boolean process_refresh_token(
   org.omg.Security.OpaqueBuffer  refresh_token
)
```

refresh_token

This parameter contains the evidence and information that may be necessary to reestablish the context.

ORBASEC SL2 currently does not make use of this operation.

discard_security_context

This operation attempts to discard the security context. It has one input parameter and one output parameter.

```
// IDL
void discard_security_context(
    in  Security::Opaque              refresh_data,
    out Security::OpaqueBuffer        out_token
);

// Java
public void discard_security_context(
   org.omg.Security.Opaque            refresh_data,
   org.omg.Security.OpaqueBufferHolder out_token
);
```

discard_data

This parameter contains the information that may be necessary to discard the context.

out_token

This parameter contains the information that is to be transmitted back to the remote side in a SECIOP DiscardContext message.

process_discard_token

This operation attempts to process a discard token produced by the **discard_security_context** operation of the remote side of the security context. It has one input parameter.

```
// IDL
boolean process_discard_token(
    in Security::OpaqueBuffer         refresh_token
);

// Java
public boolean process_discard_token(
   org.omg.Security.OpaqueBuffer   refresh_token
);
```

discard_token

This parameter contains the evidence and information that may be necessary to discard the context.

ClientSecurityContext

The **ClientSecurityContext** object is created by the **Vault** after a successful **init_security_context** operation. It is used to represent the establishment of a secure association with a target. It has the following interface:

```
interface ClientSecurityContext : SecurityContext {
  readonly attribute Security::AssociationOptions
                              association_options_used;
  readonly attribute Security::DelegationMode
                              delegation_mode;
  readonly attribute Security::Opaque
                              mech_data;
  readonly attribute SecurityLevel2::CredentialList
                             client_credentials;
  readonly attribute Security::AssociationOptions
                              server_options_supported;
  readonly attribute Security::AssociationOptions
                              server_options_required;
  readonly attribute Security::Opaque
                              server_security_name;
};
```

association_options_used

This read-only attribute states the association options that were used to make the association with the **client_credentials**. This value should be a value somewhere between the **accepting_options_required** and the **accepting_options_supported** of the **client_credentials**.

```
// IDL
readonly attribute Security::AssociationOptions
                                    association_options_used;
// Java
public short accociation_options_used();
```

delegation_mode

This read-only attribute states the delegation mode of the security context, which must be a supported delegation mode of the **client_credentials**. It stipulates that the credentials are in the a delegation mode of:

- No delegation mode (**SecDelModeNoDelegation**), where they can not be used for invocations.
- Simple delegation mode (**SecDelModeSimpleDegation**), where the credentials can be indiscriminately used on the client's behalf.
- Composite delegation (**SecDelModeCompositeDelegation**) where the credentials have some sort of composite ability, such as a trace, a combination of privileges, etc.

```
// IDL
readonly attribute Security::DelegationMode delegation_mode;

// Java
public org.omg.Security.DelegationMode
delegation_mode();
```

mech_data

This read-only attribute is the mechanism data from the IOR that was used to set up the secure association, in its raw form.

```
// IDL
readonly attribute Security::Qpaque mech_data;

// Java
public byte[] mech_data();
```

client_credentials

This read-only attribute holds the credentials object that was used to create the secure association with the target. These credentials can be either of the “own” credentials type, or “received” credentials type.

```
// IDL
readonly attribute SecurityLevel2::Credentials
                                        client_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
client_credentials();
```

Note – For internal integrity, the credentials placed in this attribute while on the **ClientSecurityContext** should be a non-modifiable copy of the credentials used to create the context, because the application can manipulate various attributes of the credentials.

server_options_supported

This read-only attribute holds the attributes that the server is said to support for the selected mechanism, i.e. from the **target_supports** attribute of the selected security component in the target’s IOR.

```
// IDL
readonly attribute Security::AsssociationOptions
                                    server_options_supported;

// Java
public short server_options_supported();
```

server_options_required

This read-only attribute holds the attribute that the server is said to require for the selected mechanism, i.e. from the **target_requires** attribute of the selected security component in the target's IOR.

```
// IDL
readonly attribute Security::AsssociationOptions
                                    server_options_required;

// Java
public short server_options_required();
```

server_security_name

This read-only attribute holds the target's security name that was used to set up the secure association.

```
// IDL
readonly attribute Security::Qpaque
                                    server_security_name;

// Java
public byte[] server_security_name();
```

Server Security Context

The **ServerSecurityContext** object is created by the **Vault** after a successful **accept_security_context** operation. It is used to represent the establishment of a secure association with a client. It has the following interface:

```
interface ServerSecurityContext : SecurityContext{
    readonly attribute Security::AssociationOptions
                                  association_options_used;
    readonly attribute Security::DelegationMode
                                  delegation_mode;
    readonly attribute SecurityLevel2::CredentialsList
                                  server_credentials;
    readonly attribute Securitye::AssociationOptions
                                  server_options_supported;
    readonly attribute Security::AssociationOptions
                                  server_options_required;
    readonly attribute Security::Opaque
                                  server_security_name;
};
```

association_options_used

This read-only attribute states the association options that were used to make the association with the **server_credentials**. This value should be a value somewhere between the **accepting_options_required** and the **accepting_options_supported** of the **server_credentials**.

```
// IDL
readonly attribute Security::AssociationOptions
                                    association_options_used;
// Java
public short accociation_options_used();
```

delegation_mode

This read-only attribute states the delegation mode of the security context, which must be the same as the delegation mode of the **received_credentials**. It stipulates that the credentials are in the a delegation mode of:

- No delegation mode (**SecDelModeNoDelegation**), where they can not be used for invocations.
- Simple delegation mode (**SecDelModeSimpleDegation**), where the credentials can be indiscriminately used on the client's behalf.
- Composite delegation (**SecDelModeCompositeDelegation**) where the credentials have some sort of composite ability, such as a trace, a combination of privileges, etc.

```
// IDL
readonly attribute Security::DelegationMode delegation_mode;

// Java
public org.omg.Security.DelegationMode
delegation_mode();
```

server_credentials

This read-only attribute holds the credentials object that was used to create the secure association with the client. This **Credentials** object should be one of the credentials objects given to **accept_security_context**.

```
// IDL
readonly attribute SecurityLevel2::Credentials
                                        server_credentials;

// Java
public org.omg.SecurityLevel2.Credentials
server_credentials();
```

Note – For internal integrity, the credentials placed in this attribute while on the **ServerSecurityContext** should be a non-modifiable copy of the credentials used to create the context, because the application can manipulate various attributes of the credentials.

server_options_supported

This read-only attribute holds the attributes that the server is said to support for the selected mechanism, i.e. from the **target_supports** attribute of the selected security component in the target's IOR that was used to set up the security context.

```
// IDL
readonly attribute Security::AsssociationOptions
                                    server_options_supported;

// Java
public short server_options_supported();
```

server_options_required

This read-only attribute holds the attribute that the server is said to require for the selected mechanism, i.e. from the **target_requires** attribute of the selected security component in the target's IOR that was used to set up the security context.

```
// IDL
readonly attribute Security::AsssociationOptions
                                    server_options_required;

// Java
public short server_options_required();
```

server_security_name

This read-only attribute holds the target's security name that was used to set up the secure association.

```
// IDL
readonly attribute Security::Qpaque
                                    server_security_name;

// Java
public byte[] server_security_name();
```

CHAPTER 9

Security Opaque Encodings

Opaque Encodings

CORBA Security Level 2 functionality has many a data structure containing **Security::Opaque** data typed elements, which is defined below as:

```
// IDL
module Security {
     typedef sequence<octet> Opaque;
};
```

The IDL to Java mapping translates this data type into a `byte[]` in Java.

The **Security::Opaque** data type is used in several places where it affects the Security Level 2 API:

- As the **security_name** parameter of the **authenticate** operation of the **PrincipalAuthenticator** object.
- As the **defining_authority** and **value** fields of the **Security::SecAttribute** structure, which is returned from a **get_attributes** operation on the **SecurityLevel1::Current** object and a **SecurityLevel2::Credentials** object.

- The **security_name** field of the **Security::SecurityMechanismData** structure, which is returned from the **get_security_mechanisms** operation on the **SecurityLevel2::Current** object.

The problem is that the “opaqueness” of these data fields are a hinderance to applying general security solutions unless you know the format or the byte encodings of all particular fields ahead of time. However, this quickly falls apart if you have two different mechanisms that can deliver different byte encodings for such things as security attributes containing an Access Id. With one mechanism it may be a straight byte to ASCII character string translation, in another it may be one of two different encodings, such as a string, or a binary X.500 Directory Name, which is used in X.509 certificates.

Unfortunately, the CORBA Security Specification is quite lacking in the respect of making sense of the “opaqueness” of security attributes. One would hope that a better scheme will develop over time. In the meantime, Adiron has developed a utility for ORBASEC SL2 containing functionality for generalizing and typing the byte encodings of such applications of the **Security::Opaqu**e data type. This utility is a class called **Opaque** with staticly defined functions.

The Opaque Class

A security name may be several different types and have several different encodings into bytes. In order to make sense of this “opaqueness”, ORBASEC SL2 introduces a utility object class called **Opaque** which resides in the **orbasec.corba** package.

This utility class encodes names of different types into a tag value and a byte encoding and packages them up in a CDR encapsulation. When names, such as security names, or the fields of the **Security::SecAttribute** structure, are used at the Security Level 2 API level, they must be in this CDR encapsulation format.

A better way to say this restriction is that any time you access a **Security::Opaque** field or parameter as a name of something, (e.g. a security name or a field of an attribute) wrap it using the **Opaque** class before encoding or decoding it to a **Security::Opaque** (i.e. byte[]).

The Opaque Interface

A brief introduction to the **Opaque** class specifying the most used features of it and how it is used is given here. However, please see the *JavaDoc* built documentation for the **orbasec.corba.Opaque** class for the more precise details.

```
public abstract class Opaque {
  // Static classes
  public static class KerberosName extends Opaque { .... }
  public static class DirectoryName extends Opaque { .... }
  public static class PrintableString extends Opaque {....}
  ....

  // Static Functions
  public static Opaque encodeKerberosName(String name);
  public static Opaque encodeDirectoryName(byte[] der_dn);
  public static Opaque encodePrintableString(String name);
  ....

  public static Opaque decode(byte[] opaque_encoding);

  // Instance Functions
  public String  toString(); // Overrides Object
  public byte[]  getEncoding();
  public byte[]  getRawBytes();
  ....
}
```

The **Opaque** class has support for the encoding and decoding of different name types to and from **Security::Opaque**, such as printable strings, X.500 binary encoded Distinguished Names, Kerberos names, and more. The entire interface is not presented here. A thorough explaination is in the *JavaDoc* built documentation for the **orbasec.corba.Opaque** class.

The Opaque.encode Methods

The encoding functions all have the form of:

```
Opaque encode<subclass name>( <parameters> )
```

They may take one or more parameters and create an object that is a subclass of the **Opaque** class. The parameters represent the content of the intended name in some form.

For example, examine the **Opaque.encodeKerberosName** method. This method takes the string that represents a Kerberos name in string form, and creates a **Opaque** object. The most common use of this object would be in a call the **PrincipalAuthenticator** object's authenticate operation. An example of this scenario follows:

```
// Java
// The Principal Authenticator comes from Security Current
org.omg.SecurityLevel2.PrincipalAuthtenticator pa = .....

// A few holders for out parameters of authenticate
org.omg.SecurityLevel2.CredentialsHolder credsh =
             new org.omg.SecurityLevel2.CredentialsHolder();
org.omg.Security.OpaqueHolder cont_datah =
             new org.omg.Security.OpaqueHolder();
org.omg.Security.OpaqueHolder auth_specific_datah =
             new org.omg.Security.OpaqueHolder();

// A normal Kerberos Name
String           principal      = “bart@MYREALM.COM”;

// The kerberos name encoded as a Opaque object
orbasec.corba.Opaque namePrincipal =
          orbasec.corba.Opaque.encodeKerberosName(principal);

// The Opaque encoded as bytes.
byte[]           security_name = namePrincipal.getEncoding();

// A call to authenticate
pa.authenticate(
    0,                                      // method
    “Kerberos”,                            // mechanism
    security_name,                          // security_name
    (“cache_name=MEMORY:0\n”+
     “password=\”mypassword\”\n”).getBytes(),// auth_data
    new org.omg.Security.SecAttribute[0],   // privileges
    credsh,                                 // creds holder
    cont_datah,
    auth_specific_datah
);
```

In the above example, you can see that the normal Kerberos name of "bart@MYREALM.COM" went through two transformations before it became a byte array suitable for use with the **PrincipalAuthenticator** object, namely

$$\textbf{\textit{String}} \rightarrow \textit{encodeKerberosName} \Rightarrow \textbf{\textit{Name}} \rightarrow \textit{getEncoding} \Rightarrow \textbf{\textit{Opaque}}$$

The Opaque.decode Operation

To get a name back from an **Opaque** encoding you must use the **Opaque.decode** function. One might do this, in order to perform access checks, i.e. you may need to do a comparison on the AccessId security attribute value.

For the following example, assume that we have retrieved the "received" credentials, the **ReceivedCredentials** object representing a client's Kerberos identity.

```
// Java
org.omg.SecurityLevel2.Current current = // get current

org.omg.SecurityLevel2.ReceivedCredentials creds =
                    current.received_credentials();

org.omg.Security.AttributeType[] attr_types =
       new org.omg.Security.AttributeType[1];

// Generate an AttributeType for AccessId
// FAMILY DEFINER 0, FAMILY 1, AccessId = 2
attr_types[0] = new org.omg.Security.AttributeType(
                     new org.omg.Security.ExtensibleFamily(
                             (short) 0, (short) 1 ),
                       2);

org.omg.Security.SecAttribute[] attrs =
       creds.get_attributes(attr_types);

try {
   Opaque kName = Opaque.decode(attrs[0].value);
    System.out.println("Access id =" + kName.toString());
} catch (Opaque.CodingException e) {
   System.out.println(e);
}
```

In the above code segment beyond all the set up for retrieving an **AccessId** security attribute, is the decoding of the **Opaque** encoding. You will notice that **Opaque.decode** may throw a coding exception. This exception is thrown if the data doesn't unmarshal correctly.

Note – All ORBASEC SL2 internal mechanisms use this Opaque utility to encode Opaque security names and fields of security attributes, so decoding should be okay.

You will also notice that other classes that were not previously shown exist for the different forms of names, such as **Opaque.KerberosName**. This class is a class that extends **Opaque**, but it is defined within the scope of the **Opaque** class. Such as:

```
public abstract class Opaque {
  ...
   public static class KerberosName extends Opaque {
       String name;
       public toString()
       {
         return name;
       }
       public byte[] getRawBytes()
       {
           return name.getBytes();
       }
   }
}
```

You might think that all of this encoding/decoding mechanisms is excessive, until you consider using SSL. The SSL protocol uses X.509 certificates in which the subject's identifier is in the form of an X.500 Directory Name. An X.500 Directory Name (DN), sometimes called "Distinguished Name", is an ASN.1 binary data structure encoded with the Distinguished Encoding Rules (DER).

In order to parse one of these names, one must have a provider that can decode the DER encoding of the ASN.1 structure representing the DN.

Note – If you have the ORBASEC SL2-SSL plug-in module, the IAIK toolkit has that functionality.

For the following example, consider the case in which the **AccessId** attribute is retrieved from a **Credentials** object using the SSL supplied protocol. This means that **value** field of the **AccessId** security attribute is an **orbasec.corba.Opaque** encoded DN.

```
try {
   // Get the Directory Name
   Opaque.DirectoryName  dName =
         (Opaque.DirectoryName) Opaque.decode(attrs[0].value);
   byte[] name = dName.name;
   try {
      // Use IAIK to parse it and turn it into a string.
      iaik.asn1.structures.Name dn =
           iaik.asn1.structures.Name(name);
       System.out.println("Access id =" + dn.toString());
   } catch (iaik.asn1.CodingException e) {
      System.out.prinln(e);
   }
} catch (Opaque.CodingException e) {
   System.out.println(e);
}
```

The class **Opaque.DirectoryName** contains a

```
byte[] name;
```

field. This field represents the raw binary structure of a X.500 Directory Name, not its **Opaque** encoding. This "raw" binary structure is the DER encoding of a DN. However, the **Opaque** class does nothing to enforce that the raw bytes are actually a DER encoding of a DN.

Other name forms are supported, such as RFC822, which is an Email name. Again, no structure is enforced; however, the **name** component for a **orbasec.corba.Opaque.RFC822Name** object is a simple **String**.

Please check the *JavaDoc* built documentation that comes with the ORBASEC SL2 distribution for more details and interfaces.

CHAPTER 10

The SL2 Class

The SL2 Class

ORBASEC SL2 has a Java class that contains statically defined methods that are used to initialize SL2. It also contains statically defined methods that help with such things like creating certain Security Level 2 policy objects. The interface for the SL2 Class is:

```
// Java
package orbasec;

public class SL2
{
  // ORBAsec SL2 Version string
  public static String Version;

  public static void init(
        String                argv[],
        java.util.Properties  properties
  );
```

```
public static void init(
       java.applet.Applet    applet,
       java.util.Properties  properties
);

public static void init_with_boa(
       String                argv[],
       java.util.Properties  properties
);

public static org.omg.CORBA.ORB
orb();

public static org.omg.CORBA.BOA
boa();

public static org.omg.SecurityLevel2.QOPPolicy
      org.omg.Security.QOP    qop
);

public static org.omg.SecurityLevel2.MechanismPolicy
create_mechanism_policy(
   String[]             mechanisms
);

public static
org.omg.SecurityLevel2.InvocationCredentialsPolicy
create_invoc_creds_policy(
    org.omg.SecurityLevel2.Credentials[]   creds_list
);

public static org.omg.SecurityLevel2.EstablishTrustPolicy
create_establish_trust_policy(
    org.omg.Security.EstablishTrust        trust
);

public static
org.omg.SecurityLevel2.DelegationDirectivePolicy
create_delegation_directive_policy(
   org.omg.Security.DelegationDirective
                                        delegation_directive
)

public static
orbasec.SecLev2.TrustedAuthorityPolicy
```

```
  create_trusted_authority_policy(
     orbasec.SecLev2.TrustedAuthorityPolicyContent
                                trusted_authorities
  );
};
```

Version

This field contains a string describing the version of ORBA_{SEC} SL2 that you are working with.

init (String parameter)

This initializer is used to initialize the security service of the ORB for stand-alone "pure client" CORBA applications. The String array and Properties parameters are passed to the ORB initialization methods. Use the **orb** accessor of this class to obtain a reference to the ORB. This method is described in detail in the chapter entitled "Initializing SL2" on page 45.

init (Applet parameter)

This initializer is used to initialize the security service of the ORB for Java Applets, which are "pure client" applications, since they cannot accept connections. The Applet and Properties parameters are passed to the ORB initialization methods. Use the **orb** accessor of this class to obtain a reference to the ORB. This method is described in detail in the chapter entitled "Initializing SL2" on page 45.

init_with_boa

This initializer is used to initialize the security service of the ORB for stand-alone CORBA applications that are capable of accepting connections, i.e., act as CORBA servers. The String array and Properties parameters are passed to the ORB and BOA initialization methods. Use the **orb** and **boa** accessors of this class to obtain a reference to the ORB and BOA, respectively. It is described in detail in the chapter entitled "Initializing SL2" on page 45.

orb

This accessor returns a reference to the **org.omg.CORBA.ORB** initialized in one of the above SL2 initializers.

boa

This accessor returns a reference to the **org.omg.CORBA.BOA** initialized in the **init_with_boa** initializer. If ORBASEC SL2 was not initialized with the **init_with_boa** initializer, this accessor returns `null`.

create_qop_policy

This operation is a convenience function that acts as a factory for creating a simple quality of protection policy, i.e. a **QOPPolicy** object.

Note – This function does not preclude the implementation of a more complicated policy, such as depending on the time of day, location, or other environmental considerations.

create_mechanism_policy

This operation is a convenience function that acts as a factory for creating a simple mechanisms policy that stipulates the mechanisms to be used during invocations on targets, e.g. a **MechanismPolicy** object.

Note – This function does not preclude the implementation of a more complicated policy, such as depending on the time of day, location, or other environmental considerations.

create_invoc_creds_policy

This operation is a convenience function that acts as a factory for creating a simple invocation credentials policy, e.g. an **InvocationCredentialsPolicy** object. However, the credentials list given as input to this function should be a valid credentials list for an **InvocationCredentialsPolicy** object.

Note – This function does not preclude the implementation of a more complicated policy, such as depending on the time of day, location, or other environmental considerations.

Actually, ORBASEC SL2 has two policies that can be used out of the box. They are defined by the attributes:

- `orbasec.SL2.OwnInvocationCredentialsPolicy`

Always returns the **own_credentials** attribute of Security **Current** when policy analysis is performed at binding time.

- `orbasec.SL2.ReceivedInvocationCredentialsPolicy`

  Always returns the **received_credentials** attribute of Security **Current** in a single element list when policy analysis is performed at binding time.

create_establish_trust_policy

This operation is a convenience function that acts as a factory for creating a policy that stipulates whether client and/or target authentication should be established during an invocation, e.g. an **EstablishTrustPolicy** object. ["Establish Trust Policy" on page 132].

Note – This function does not preclude the implementation of a more complicated policy, such as depending on the time of day, location, or other environmental considerations.

create_delegation_directive_policy

This operation is a convenience function that acts as a factory for creating a policy that stipulates whether the credentials being used for the invocation should be delegated to the target or not, e.g. an **DelegationDirectivePolicy** object. ["Delegation Directive Policy" on page 132].

Note – This function does not preclude the implementation of a more complicated policy, such as depending on the time of day, location, or other environmental considerations.

create_trusted_authorities_policy

This operation is a convenience function that acts as a factory for creating a policy that lists the authorities that are trusted for authentication. [see "TrustedAuthorityPolicy" on page 138].

CHAPTER 11

Other Java Utility Classes

Other Java Utility Classes

ORBASEC SL2 has a number of Java Utility Classes that contain statically defined functions that help with certain aspects of dealing with CORBA Security Level 2 interfaces and Java in general, which the internals of ORBASEC SL2 actually use.

The utilities come in the following Java packages:

orbasec.util. This package contains classes for implementing a Linked List utility, debugging, and some functions for printing hexadecimal buffers, etc. that are actually used by ORBASEC SL2.

orbasec.io. This package contains some Input/Output classes for manipulating files, and other general manipulating **java.io** objects.

orbasec.corba. This package contains classes that help with CORBA and the Security Interfaces.

orbasec.tools. This package contains some stand-alone tools for dealing with certain external aspects of the system. For example, if you have the ORBASEC SL2-SSL distribution, this package contains a tool for generating simple X.509 certificates. See the JavaDoc generated documentation that comes with your ORBASEC SL2 distribution for command syntax of these tools.

The obvious classes of interest to the ORBASEC SL2 user are in the **orbasec.corba** package. Some of the most important classes are listed below.

Class	Purpose
Opaque	This class contains classes and functions for creating Opaque encodings for the content of security names and security attribute values. See "Opaque Encodings" on page 185.
MechUtil	This class contains statically defined fields and functions that deal with the Kerberos and SSL mechanism strings. Its best use is already defined strings that represent the available cipher suites and mechanisms in ORBASEC SL2.
CredUtil	This class contains statically defined functions for querying **SecurityLevel2:Credentials** objects, creating security attribute types and security attributes, printing out credentials, etc.
AttrDef	This class contains statically defined constants for attribute type and family definers used by CORBA and Adiron. It also contains convience functions for constructing SecAttribute structures.
IOPUtil	This class contains statically defined functions for querying and manipulating IORs.
CDRBuffer CDRDecoder CDREncoder TypeCode	Lightweight CDR encoders and decoders that implement **org.omg.CORBA.portable.InputStream** and **org.omg.CORBA.portable.OutputStream** interfaces of the IDL/Java mapping. These classes do not handle complex data types such as "any" or recursive data types.
U	A utility for doing translations between org.omg.SECIOP.ulonglong structures and the Java long primitive type.

TABLE 13. Some Members of the orbasec.corba Package

Documentation for these utilities can be found in the JavaDoc generated explanations that can be found in the documentation API section of your ORBASEC SL2 distribution.

CHAPTER 12

References

1. Kohl J, Neuman C., "The Kerberos Network Authentication Service (V5)", Network Working Group RFC 1510, September 1993.
2. The Object Management Group, "The Common Object Request Broker: Architecture and Sepcification", Version 2.2, Feburary 1998.
3. The Object Management Group, "CORBAservices: Common Object Services Specification", November 1997.
4. The Object Management Group, "Security Service Specification", Version 1.2 Draft 4.1, 5 January, 1998.
5. The Object Management Group, "Security Service Specification", Version 1.5, March 1999.
6. Object Oriented Concepts, Inc. "ORBACUS C++ and Java", Version 3.1, 1999.