ORBASEC SL2

User Guide

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

Version 2.1.4
July 2000

CHAPTER 1

ORBASEC SL2 Introduction

What is ORBASEC SL2?

THIS IS THE RELEASE 2.1 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 adopted 1.7 Revision of the Security Service Specification [6].

The features ORBASEC SL2 supports:

- Full functionality of the ORB, ORBACUS 3.3 for Java.[9]
- CORBA Security Level 2 Functionality.
- SECIOP - SECure Inter-Operability Protocol compliant.
- Security Replaceability
- Kerberos Version 5 (GSS-API)
- Secure Sockets Layer Version 3 (SSLv3)
- Unprotected Communication (IIOP)

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

Developments Since ORBASEC SL2 2.0

Since ORBASEC 2.0, we have reorganized the interfaces to CORBA Security objects to conform with changes made to the specification as of Security RTF 1.7. Some of these changes are not backwards compatible with previous versions of ORBAsec SL2, so you should read RTF 1.7 to see what changes, if any, effect your code. Any changes that are backwards compatible have been made deprecated features. Adiron makes no guarantee about how long these deprecated features will remain a part of ORBAsec SL2, so developers should make every effort to remove any dependencies in their own code on these deprecated features.

We have also improved many features of ORBASEC SL2 in response to user feedback. These improvements include an enhancement of SL2 initialization, allowing the use of the standard **CORBA::ORB** initialization procedures supplied by the ORBACUS ORB, implementation of a host of authentication methods for the GSS-Kerberos, SSL, and IIOP security mechanisms, and improved methods of supplying authentication data for authentication SSL credentials using the Sun Java Cryptography Architecture.

What is ORBACUS?

ORBASEC SL2 is implemented on top of ORBACUS 3.3 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 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[6] that gives a certain level of functionality to the application programmer in the form of an API. Its basic features are:

- Security Manager Object
- 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 programmer, 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 161.

What is Kerberos?

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

ORBASEC SL2-GSSKRB distribution comes with a Security Replaceable Module supporting the GSS-Kerberos from the Massachusetts 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[2] 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, **SSL_IAIK.jar**.

The ORBASEC SL2-SSL distribution gives the application developer 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, though application developers may use ORBAsec SL2 to provide their own integration with PKI components. If your organization is in need of custom integration with a PKI, please contact Adiron for more information.

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 or JDK 1.2 from JavaSoft, Inc.
- ORBACUS 3.3 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 or greater, from the Athena Project at Massachusetts Institute of Technology. Our Kerberos libraries are based on the 1.1.1 release.

ORBASEC SL2-SSL requires a license from the following:

- iSaSiLk 2.51 from IAIK, the Institute for Applied Information Processing and Communication, Graz, University of Technology, Graz, Austria. Since the IAIK SSL libraries require IAIK-JCE 2.51, IAIK's implementation of the Java Cryptography Extension Provider interfaces, you will need these libraries, as well.

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 written 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, authentication, encryption, or the Kerberos protocol in general.

Books on CORBA Security

One of the only books out on CORBA security at the moment is the CORBA Security Specification itself.[6] 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.

Another book on CORBA Security has been recently released. It is:

Blakely, Bob, “CORBA Security: An Introduction to Safe Computing with Objects”, The Addision-Wesley Object Technology Series, ISBN: 0201325659, October, 1999.

Bob Blakely is one of the many important cooperating authors of the CORBA Security Specification.

The Future of ORBASEC SL2

ORBASEC SL2 is not a fully compliant Security Level 2 implementation, but it is fully conforms to the specification. A fully compliant Security Level 2 implementation cannot be implemented as a single ORB library, it must be an enterprise solution involving other running 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. ORBAsec SL2 provides the IDL for these interfaces.

The purpose behind ORBAsec SL2 is to provide first and foremost, authentication, integrity, and confidentiality services for applications. Therefore, it is very likely that applications built with the bare ORBAsec SL2 will be “security aware” applications.

Adiron has recently released a product that is built with ORBAsec SL2 that provide automatic authentication and access control edging toward the goal of securing CORBA applications that are not “security aware”. This product is called CONTROL and it extends ORBAsec SL2. Please see the Adiron web site for more details on the way you can make use of automatic authentication and access control and keep your application code “security unaware”.

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, the ORB which ORBAsec SL2 extends, does not implement them. These interfaces pertain to domain management, which is security related.

- Construction Policy Object
- Domain Manager Object

It is likely that in the upcoming submissions to the Domain Management RFP from the OMG that these objects will be deprecated.

Adiron, LLC is still 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 (including JDK 1.2)
- Object Oriented Concepts ORBACUS 3.3 for Java
- An Operational MIT Kerberos 5, version 1.0.5 or greater 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 Java Development Kit (JDK), version 1.1.6 or later (including JDK 1.2)
- Object Oriented Concepts ORBACUS 3.3 for Java
- IAIK's iSaSiLk 2.51 SSL libraries and IAIK-JCE 2.51 toolkit

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 several demonstration programs for you to experiment with. Two of these demos are a modification of the ORBACUS "Hello World" application, which is a simple distributed application based on a familiar introductory programming example. One of the demos, in the krb-hello directory, uses the GSS-Kerberos plug-in for using Kerberos autentication and encryption. The other demo, ssl-hello, is functionally identical to the krb-hello demo, but uses the SSL plug-in for SSL authnetication and encryption. The source files `Server.java`, `Hello_impl.java`, and `Client.java` have been modified in both of these demos to illustrate a few of the simple but powerful security features ORBASEC provides.

In this chapter, we will walk through the provided source code for the client and server programs in both demos, 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 [9].

To run the krb-hello demo application in this chapter, you should have

1. The files **SL2.jar** and **GSSKRB.jar** in your **CLASSPATH** in front of **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 principa 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 Kerberos password authentication in ORBASEC SL2.

Note – If you use the service-principal/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.

To run the ssl-hello demo application in this chapter, you should have

1. The files **SL2.jar** and **SSL_IAIK.jar** in your **CLASSPATH** in front of **OB.jar** and IAIK jar files (**iaik_ssl.jar** and **iaik_jce.jar**).

Note – If you are using JDK 1.2, you must install the JCE cryptographic libraries as a Java extension. Consult your local Java documentation for instructions about how to install third-party security “providers” with your Java installation.

The private keys and X509 Certificates required for authentication are provided in *demo.keystore* file that comes with the ssl-hello demo program. This keystore file uses the IAIK keystore file format, so you will need the IAIK JCE cryptographic libraries to use this keystore file in the demo. You may use your own keystore, but you will need to modify the authentication parameters described below. See Figure , “KerberosServiceClient Authentication Method,” on page 105 for details.

Adiron’s Test Kerberos Key Distribution Center (KDC)

For Kerberos users, if you use the supplied Kerberos configuration file, `orbasec_krb5.config`, this directs your Kerberos configuration to the KDC on line at Adiron, allowing hosts with unrestricted Internet access to run the demonstration programs out-of-the-box. This KDC is for your use in evaluation of ORBASEC SL2-GSSKRB; however, Adiron can make no guarantees that it will remain on-line indefinitely, that it will remain in the same location, or that the principals 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.

Using the JDK 1.2

The JDK 1.2 platform comes with some annoying restrictions of which there are workarounds.

The first is the separation of the CLASSPATH into a user section and a system section. The system section is commonly known as the "bootclasspath". Classes found in the bootclasspath cannot be overriden by classes in the user classpath. This procedure seems sensible so that you cannot override key Java components. However, Sun installed a "client only" ORB into the jar file, making it impossible to use other ORBs with this library. Therefore, if you are using the JDK 1.2, you need to use the "-Xbootclasspath" extended flag when starting up a client or server to include ORBAcus and ORBAsec SL2 in front of the "rt.jar". If you are using a "C-shell" or "tcsh" we recommend setting up your CLASSPATH environment variable as you would with the JDK 1.1 to include **SL2.jar**, **GSSKRB.jar**, **SSL_IAIK.jar**, **OB.jar**, **rt.jar** in that order, and also set up the following alias for the **java** command.

```
alias java java -Xbootclasspath:\$CLASSPATH
```

Notes on Kerberos and use of the JNI with the JDK 1.2

Another restriction that the JDK 1.2 presents us with is that classes in the bootclasspath that call the **System.loadLibrary** function, which loads a Java Native Interface (JNI) libraries, need to have those shared library files (**.so** files on unix, **.DLL** files on NT) in the JDK's system library directory. **No user flag exists to override this restriction!** Since the Kerberos version of ORBAsec SL2 uses the JNI, you may have to get your adminstrator to install the following files that are found in the ORBAsec SL2 **lib** directory into the specified directories below:

TABLE 1. JNI File Locations for the JDK 1.2

Platform	Files	Location
Linux	libJgsskrb5.so	$JAVA_HOME/jre/lib/i386/

TABLE 1. **JNI File Locations for the JDK 1.2**

Platform	Files	Location
Solaris	libJgsskrb5.so	$JAVA_HOME/jre/lib/sparc
NT	libJgsskrb5.DDL krb5_32.DLL k5crypto.DLL gssapi_32.DLL	%JAVA_HOME%/jre/bin

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 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();
}
```

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.

Hello Implementation

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);
    }
  }
}
```

When the **hello** method of a **Hello** object is invoked, the ORB is asked for a reference to a **SecurityLevel2::Current** object; this object is used to obtain *thread specific* information about the current security context. The **SecurityLevel2::Current** interface is discussed in deatil in “Security Current” on page 79. In this case, we ask for the received credentials, the **SecurityLevel2::ReceivedCredentials** object of the invoking client received via the secure association established. The **ReceivedCredentials** interface is discussed in detail in “Received Credentials” on page 139.

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

Server Implementation

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 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 command-line options in the args parameters, together with any user-supplied **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 static 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::SecurityManager** object, from which we will obtain security-related functionality for the **Server**:

```
// Get SecurityLevel2::Current from ORB
org.omg.CORBA.Object obj =
  orb.resolve_initial_references(“SecurityManager”);
org.omg.SecurityLevel2.SecurityManager security_manager =
  org.omg.SecurityLevel2.SecurityManagerHelper.narrow(obj);
...
```

The **SecurityLevel2::SecurityManager** interface is discussed in detail in “SecurityManager” on page 69.

Note – Unlike **SecurityLevel2::Current**, the **SecurityLevel2::SecurityManager** object holds *“capsule”* (or process) specific security information, information that is not specific to a particular thread of execution. Like the **SecurityLevel2::Current** object, it is only available on the ORB after ORBASEC SL2 has been initialized.

With a reference to the **SecurityLevel2::SecurityManager**, we can obtain the **PrincipalAuthenticator**, the **SecurityLevel2** object we use to initialize the **Server**’s credentials.

```
// Obtain PrincipalAuthenticator from SecurityManager
org.omg.SecurityLevel2.PrincipalAuthenticator pa;
pa = security_manager.principal_authenticator();
```

The **SecurityLevel2::PrincipalAuthenticator** interface is discussed in detail in “Principal Authenticator” on page 87.

Server Kerberos Authentication

To authenticate using Kerberos, we supply the “Kerberos” identifier as the **mechanism** parameter to the **PrincipalAuthenticator**’s **authenticate** method. We must also supply a valid Kerberos principal in the **security_name**, and in addition, mechanism-specific authentication data:

```
String mechanism     = “Kerberos”;
String security_name = “homer@MYREALM.COM”;
```

ORBASEC SL2 offers numerous authentication methods for use with the GSS-Kerberos security mechanism. The authentication methods are discussed in detail in “Kerberos Authentication Methods and Data” on page 95. In this example, we illustrate the Kerberos password authentication method.

To use ORBASEC SL2 authentication methods, you must supply a **Security::AuthenticationMethod** (an **int**, under the Java mapping), together with a **CORBA::Any** which contains a CORBA data structure compatable with the specified authentication method. In this case, we use the **SecLev2::SecKerberosPassword** authentication method, with a **SecLev2::KerberosPassword** data structure. This authentication method and data structure are defined in detail in “Kerberos-Password Authentication Method” on page 95.

```
int method = orbasec.SecLev2.SecKerberosPassword.value;
orbasec.SecLev2.KerberosPassword data =
    orbasec.corba.AuthUtil.default_KerberosPassword();

data.config   = "FILE:orbasec_krb5.conf";
data.password = "mypassword";
data.usage    =
    orbasec.SecLev2.CredentialsUsage.SecAcceptOnly;

org.omg.CORBA.Any auth_data = orb.create_any();
orbasec.SecLev2.KerberosPasswordHelper.insert(
    auth_data, data );
```

We use the **orbasec.corba.AuthUtil default_KerberosPassword** factory method to create a default instance of a **SecLev2::KerberosPassword** structure. Doing so ensures that the structure is marshallable into a **CORBA::Any**, and that any default values ORBASEC SL2 expects are defined. We then use the ORB to create an instance of an **Any**, and then use th **KerberosPasswordHelper** class to insert the structure.

Once the security name, security mechanism, authentication method and authentication data are set, we can call the **authenticate** operation of the **PrincipalAuthenticator**. with the follwoing additional values.

```
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 **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”`).

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 **CORBA::Any** containing information used for the GSS-Kerberos Security Mechanism.

The above example speciies that:

- `orbasec_krb5.conf` is the configuration file that states where the KDC resides;
- The principal is authenticated with the password `“mypassword”`.
- That the Credentials, once established, may only be used to accept secure invocations from clients.

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::SecurityManager** object’s **own_credentials** list 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.

Server SSL Authentication

To authenticate using SSL, we supply the “SSL” identifier as the **mechanism** parameter to the **PrincipalAuthenticator**’s **authenticate** method. We do not, however, provide a security_name; the name of the principal is obtained through the keystore file specified in the mechanism-specific authentication data.

Note – The keystore file is stored on the local file system. However, it is password protected, so the private key is encrypted.

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

int    method = 0;
String mechanism = “SSL”;
String security_name = “”;
byte   auth_data[] =
         (“format=keystore\n” +
          “keystore=FILE:./demo.keystore\n” +
          “keystore.storetype=IAIKKeyStore\n“ +
          “keystore.storepass=blahblah\n“ +
          “keystore.alias=server\n“ +
          “keystore.keypass=blahblah\n“ +
          “enable_server=true\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 `SSL` mechanism).

The **security_name** parameter indicates the principal name to be recognized by the specified security mechanism. In this case, we provide an empty string, since the name principal's name is obtained from the X509 Certificate specified in the auth_data parameter.

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

The above properties specify that:

- the principal's private key and certificate are to be obtained using the Java KeyStore facility;
- The name of the keystore file is *demo.keystore*, in the current working directory;
- The type of the keystore is IAIKKeyStore. Omit this property if you would like to use the default keystore type (JKS on must systems; consult your systems administrator for details about your default installation);
- The password used to verify the contents of the keystore is "blahblah";
- The principal's private key and X509 Certificate is obtained from the keystore via the alias "server".
- The principal's private key is locked with the password "blahblah" (*coincidentally* the same as the storepass).

- The credentials should be initialized to serve as “accepting” credentials, as well as (ordinary) invocation credentials. The enable_server property must be set to true in order to allow SSL Credentials to function as accepting Credentials.

Note – All of the definable SSL properties and their meanings are given in [“SSL Authentication Methods and Data” on page 109].

The ORBASEC SL2 implementation of SSL imposes the restriction that all server applications must have the *enable_server* property in the **auth_data** parameter set to true.

The **privileges** parameter specifies privileges that must be authenticated through the security mechanism. The SSL 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::SecurityManager** object’s **own_credentials** list 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 SSL 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 the ORBACUS demo; 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 the ORBACUS demo, 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::SecurityManager**.

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

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

  // Get SecurityLevel2::SecurityManager from ORB
  org.omg.CORBA.Object obj =
    orb.resolve_initial_references(“SecurityManager”);
  org.omg.SecurityLevel2.SecurityManager security_manager =
    org.omg.SecurityLevel2.SecurityManagerHelper.narrow(obj
  // Obtain PrincipalAuthenticator
  org.omg.SecurityLevel2.PrincipalAuthenticator pa;
  pa = security_manager.principal_authenticator();
```

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

Client Kerberos Authentication

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

```
int     method = 0;
String  mechanism = “Kerberos”;
String  security_name = “bart@MYREALM.COM”;
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”`)..

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 ["Kerberos Authentication Methods and Data" on page 95], 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 59 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 `""` 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.

Note – 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.

Client SSL Authentication

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

```
int     method = 0;
String  mechanism = “SSL”;
String  security_name = “”;
byte    auth_data[] =
         (“format=keystore\n” +
          “keystore=FILE:./demo.keystore\n” +
          “keystore.storetype=IAIKKeyStore\n” +
          “keystore.storepass=blahblah\n“ +
          “keystore.alias=client\n“ +
          “keystore.keypass=blahblah\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 SSL mechanism).

The **security_name** parameter indicates the principal name to be recognized by the specified security mechanism. In this case, we provide an empty String.

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

The above properties specify that:

- the principal's private key and certificate are to be obtained using the Java KeyStore facility;
- The name of the keystore file is *demo.keystore*, in the current working directory;
- The type of the keystore is IAIKKeyStore. Omit this property if you would like to use the default keystore type (JKS on must systems; consult your systems administrator for details about your default installation);
- The password used to verify the contents of the keystore is “blahblah“;
- The principal's private key and X509 Certificate is obtained from the keystore via the alias “client“.
- The principal's private key is locked with the password “blahblah“ (*coincidentally* the same as the storepass).

Note – All of the definable SSL properties and their meanings are given in [“Kerberos Authentication Methods and Data” on page 95], and the exact values of these properties will vary according to your Kerberos 5 configuration.

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 an explanation of this code).

Note – The **Client** should be authenticated via the **PrincipalAuthenticator** *after* the ORBASEC SL2 has 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` or `sl2/demo/ssl-hello` directory, run the command:

```
make
```

You need to make sure that you have the ORBACUS jidl command in your execution path. For the Kerberos demo, you must have OB.jar, SL2.jar, GSSKRB.jar in your Java CLASSPATH; for the SSL demo, you must have OB.jar, SSL-IAIK.jar, and the IAIK SSL and JCE toolkit jarfiles in your CLASPATH.

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
```

For the Kerberos demo, you should see something close to 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,<empty>,"30514")
     SecAttribute(41244,1,1,<empty>,"128.230.99.3")
     SecAttribute(41244,1,1,<empty>,"krbtgt/MYREALM.COM@MYREALM.COM")
     SecAttribute(41244,0,0,<empty>,"Kerberos_MIT")
     SecAttribute(0,1,2,<empty>,"homer@MYREALM.COM")

Hello Server is Ready.
```

(For the SSL demo, you will see similar output, but the mechanism names and values of the SecAttributes will be slightly different, reflecting the contents of the *demo.keystore* keystore file.)

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

A Credentials objects contain a set of **Security::SecAttribute**s, used to specify additional security information over and above that provided through the Credentials interface. Note how these attributes are displayed when the Credentials are printed to the screen.

The **Security::SecAttribute** structure contains three numbers that are used to identify the attribute's type, a *Family Definer,* a *Family,* and a *Family Attribute Type.* A Family Definer is generally granted to an organization (such as the OMG, or Adiron), and a Family can be used to categorize types, each of which can be distinguished by a Family Attribute Type.

Looking closely at the above display of the Server's Credentials, notice that the last attribute listed is the AccessId attribute; this attribute has Family Definer 0 (for CORBA), Family 1 (a family of types established by the OMG), and a Family Attribute Type 2 (for AccessId). This attribute provides the access id (login, if you will) of the principal these Credentials belong to. The OMG defines many other types for security attributes (e.g., a group id or a privilege), but these types are not used in the ORBASEC demos. For more information about Security Attributes and the numbers used to identify their types, please refer to the Security Service Specification.

In addition to the security attribute types defined by the OMG, Adiron has its own family of security attributes, identified by the Family Definer value 41244, or in hexadecimal notation, 0xA11C. The Adiron attribute types used in this demo are designated as follows.

Family 0 Family Attribute Type 1 designates 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 address and port numbers will have different values than those printed here when you run the programs, but the numbers used to identify Attribute types will be the same.

In addition to the numbers that designate an attribute type, the **Security::SecAttribute** has a *defining authority* and a *value* field, both "opaque" (i.e., byte array) values. A security attribute's defining authory represents an authority which defines the encoding of the value field of the security attribute. An <empty> value indicates that the defining authority is the OMG, and that subsequently the value field is encoded using UTF-8 character encodings. The value field itself is interpreted according to the attribute type. See the Security Specification and Security RTF 1.7 for details about these parameters.

Note – Currently, all Adiron Security Attributes use UTF-8 encodings for their value fields; for consistency with OMG Security Attributes, all Adiron Security Attributes have empty defining authority fields.

Running the Client

To run the Client, type:

```
java hello.Client
```

For the Kerberso demo, you should see something close to 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,<empty>,"30515")
     SecAttribute(41244,1,1,<empty>,"128.230.99.3")
     SecAttribute(41244,1,1,<empty>,"krbtgt/MYREALM.COM@MYREALM.COM")
     SecAttribute(0,0,0,<empty>,"Kerberos_MIT")
     SecAttribute(0,1,2,<empty>,"bart@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,<empty>,"30514")
     SecAttribute(41244,1,3,<empty>,"128.230.99.3")
     SecAttribute(41244,1,2,<empty>,"30515")
     SecAttribute(41244,1,1,<empty>,"128.230.99.3")
     SecAttribute(41244,1,1,<empty>,"krbtgt/MYREALM.COM@MYREALM.COM")
     SecAttribute(41244,0,0,<empty>,"Kerberos_MIT")
     SecAttribute(0,1,2,<empty>, "bart@MYREALM.COM")
Enter 'h' for hello or 'x' for exit:
>
```

(For the SSL demo, you will see similar output, but the mechanism names and values of the SecAttributes will be slightly different, reflecting the contents of the *demo.keystore* keystore file.)

You will notice, in contrast to the Server principal’s credentials, that since the principal was authenticated as a pure client, the **accepting_options_supported** field is empty, indicating that these credentials cannot be used to accept a secure association. Notice also that the AccessId security attribute reflects the name of the principal (“bart@MYREALM.COM”, in the GSS-Kerberos demo, and ““ in the SSL demo) authenticated by the Client.

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,<empty>,”30515”)
     SecAttribute(41244,1,3,<empty>,”128.230.99.3”)
     SecAttribute(41244,1,2,<empty>,”30514”)
     SecAttribute(41244,1,1,<empty>,”128.230.99.3”)
     SecAttribute(41244,1,1,<empty>,”krbtgt/MYREALM.COM@MYREALM.COM”)
     SecAttribute(41244,0,0,<empty>,”Kerberos_MIT”)
     SecAttribute(0,1,2,<empty>, ”bart@MYREALM.COM”)
```

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**, and **delegation_state**. The **delegation_mode** indicates the delegation ability of these credentials. Here, it is no delegation, as expected; recall that we set delegation to false in the auth_data parameter when we authenticated the Client. 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, as well as **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 controlling properties of the secure association negotiated between client and server. In this section, we provide a few modifications to the "Hello World" application to illustrate some of 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*, *EstablishTrustInTarget*, 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 2 on page 43 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

TABLE 2. GSS-Kerberos Default Server Accepting Options

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

TABLE 2. 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, by whatever means it has it its disposal. (In this case, we use the underlying Kerberos security mechanism.)

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 =
    security_manager.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,<empty>,"30514")
     SecAttribute(41244,1,1,<empty>,"128.230.99.3")
     SecAttribute(41244,0,0,<empty>,"Kerberos_MIT")
     SecAttribute(41244,0,0,<empty>,"krbtgt/MYREALM.COM@MYREALM.COM")
     SecAttribute(0,1,2,<empty>,"homer@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 on the Server 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,EstablishTrustInTargett,EstablishTrustI
nClient,NoDelgation]
    delegation_mode               = SecDelModeNoDelegation
    delegation_state              = SecInitiator
    2 Security Attributes: (definer,family,type,def_auth,value)
     SecAttribute(41244,1,4,<empty>,"30514")
     SecAttribute(41244,1,3,<empty>,"128.230.99.3")
     SecAttribute(41244,1,2,<empty>,"30515")
     SecAttribute(41244,1,1,<empty>,"128.230.99.3")
     SecAttribute(41244,1,1,<empty>,"krbtgt/MYREALM.COM@MYREALM.COM")
     SecAttribute(41244,0,0,<empty>,"Kerberos_MIT")
     SecAttribute(0,1,2,<empty>, "bart@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 “published” 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 153 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 48.

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 3. 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 [4]. For more details on these specific policies see Chapter “Policies” on page 153.

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 [6] 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

The **orbasec.SL2** class provides a collection of static initialization methods for initializing a secure ORB. Some of these methods initialize the ORB and optionally the BOA automatically, so you should carefully read this chapter and make sure not initialize the ORB or BOA before calling these initialization methods.

ORBAsec ORB Initialization

Beginning with ORBASEC SL2 2.1, you may now use the standard **org.omg.CORBA.ORB.init** static method to initialize ORBASEC SL2. This capability allows you to initialize the full ORB or the ORB for Java Applets and the BOA in the standard way. The way you configure ORBAsec SL2 is through the arguments and properties, which is explained in the following chapter.

To get this capability it required to make a minor patch to ORBACUS 3.2. Therefore, you must make sure the **SL2.jar** file is *before* the **OB.jar** file in your classpath as SL2.jar contains a slightly modified com.ooc.CORBA.ORB class.

If you use the JDK's (or some other vendor's) org.omg.CORBA.ORB class to load and intialize the ORB, you must set the value of the **org.omg.CORBA.ORBClass** property to have the value "orbasec.ORB". You may set this property explicitly in the Java properties parameter to the ORB’s init method, or you may set it via Java System Properties (or, equivalently, via the -D option at the Java command line.

```
// Java
java.util.Properties props = new java.util.Properties();
props.put( “org.omg.CORBA.ORBClass”, “orbasec.ORB” );
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init( args, props );
org.omg.CORBA.BOA boa = orb.BOA_init( args, props );
```

ORBAsec SL2 comes with its own “org.omg.CORBA.ORB” class with the proper initialization, so that setting the property explicitly is not needed for Java applications.

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 ORBASEC SL2 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 ORBASEC configuration file, use the “-*ORBconfig”* command-line option.

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 ORBASEC SL2 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 63 for more ORBASEC SL2 properties.

orbasec.seciop

This property determines whether the ORBASEC SL2'S 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 SL2 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 on Unix systems need “root” privilege.

---

orbasec.ssliop

This property determines whether the ORBASEC SL2’S 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 SL2 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 on Unix systems need “root” privilege.

orbasec.ssliop.exportable_only

This property states whether only exportable encryption 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 SL2’S 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 SL2 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 SL2 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 on Unix systems 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 explicitly created in application code.

Legal Values

true	Create Kerberos Credentials with the default Kerberos session credentials cache.
false	Do not create Kerberos Credentials.

Default Value
false

orbasec.ssl_anonymous

Setting this property to **true** automatically creates an anonymous SSL **Credentials** object during ORBASEC SL2 initialization. No certificate file is required for anonymous SSL credential initialization. It uses the Anonymous Diffe-Hillman cipher suites.

Legal Values

true	Create anonymous SSL credentials
false	Do not automatically create anonymous SSL credentials.

Default Value
false

Note – The **orbasec.ssliop** property must *not* be set to **disable** in order for this property to have any effect. Furthermore, if the **orbasec.ssliop** property is set to **server**, you must initialize ORBASEC SL2 with the **orbsec.SL2 BOA_init** (or equivalently, the **init_with_boa**) method, if you initialize SL2 directly, or the ORB's **BOA_init** method, if you use the ORB initialization methods discussed above, in order to automatically acquire anonymous SSL credentials.

orbasec.allow_iiop

Setting this property to **true** automatically creates an IIOP **Credentials** object during ORBASEC SL2 initialization.

This Credentials object must be created to enable IIOP communication over the CORBA standard protocol.

Legal Values

true	Create IIOP Credentials
false	Do not create IIOP Credentials

Default Value
false

Note – The **orbasec.iiop** property must *not* be set to **disable** in order for this property to have any effect. Furthermore, if the **orbasec.iiop** property is set to **server**, you must initialize ORBASEC SL2 with the **orbsec.SL2 BOA_init** (or equivalently, the **init_with_boa**) method, if you initialize SL2 directly, or the ORB's **BOA_init** method, if you use the ORB initialization methods discussed above, in order to automatically acquire IIOP credentials.

You may use the **orbasec.ORB.get_properties** method to obtain the ORBASEC SL2 properties in effect during initialization.

```
public static java.util.Properties
get_properties();
```

Properties defined through an ORBASEC SL2 configuration file, the Java Properties argument to any of the initializers, Java System Properties, and the command line (see below) will show up in the returned Java Properties object. If any of the **BOA_init** initialization methods have been called, they will be merged into the Properties passed to the corresponding **init** method.

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 SL2 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 4, 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 4. ORBASEC command-line options

corresponding ORBAsec property.

ORBASEC SL2 also provides a **orbasec.ORB.filter_options** class method, analogous to the **com.ooc.CORBA.ORB filter_options** member method.

```
public static String[]
filter_options( String args[] );
```

This method returns the result of removing all ORBASEC SL2 command line options from the supplied list of options. It is useful for applications that rely on a specific argument syntax.

Secure 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.ORB" 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)
{
  Properties props = new Properties();
  ORB orb =
       ORB.init_with_boa( args, props) );
  BOA boa = orb.BOA_init( args, props );

  // authenticate
  ...

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

ORBASEC SL2 Security Services

During the initialization process a few services are created and are added automatically as initial services on the ORB. These services are obtainable using the ORB's resolve_initial_references method, supplying a String parameter used to

ORBAsec SL2 currently installs three such secure services, a SecurityManager, a SecurityCurrent, and a PolicyCurrent Object. These security services are discussed in “Security Services” on page 69.

We illustrate how to obtain a reference to any one of these security services with the SecurityManager object:

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

  org.omg.CORBA.Object obj =
    orb.resolve_initial_references(“SecurityManager”);

  org.omg.SecurityLevel2.SecurityManager security_manager =
    org.omg.SecurityLevel2.SecurityManagerHelper.narrow(obj);

  ...
}
```

Note – A reference to these security services 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. ORBASEC SL2 allows application programmers to provide their own **SecurityReplaceable** security components and use them with ORBASEC SL2, allowing pluggable security mechanism components within ORBASEC SL2. Please note, however, that the SecurityRelaceable module was only designed for interoperability with SECIOP, the SECure InterOperable Protocol. There is no corresponding notion of replaceability with the SSLIOP module Adiron furnishes.

Assuming you have written your own implementation of the **SecurityReplaceable** components, 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 ORB initialization. The **properties** parameter will be derived from properties established during initialization. See See “ORBAsec SL2 Configuration” on page 54. for a description about the precedence rules 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 SL2 "reserved" mechanism 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**.

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.

The SL2 Initialization Methods

The **orbasec.SL2** class provides a large collection of static methods for initializing ORBASEC SL2 which remain for backwards compatability with older versions of ORBAsec SL2.

Note – Some of these initialization methods have been added to make them more closely conform to standard forms of ORB and BOA initialization. However, we prefer you now use ORB::init and ORB::BOA_init to initialize ORBAsec SL2 as the SL2 class methods may be phased out in the future.

To initialize the ORB with ORBASEC SL2 security for standalone applications, use

```
public static org.omg.CORBA.ORB
init(
    String               argv[],
    java.util.Properties properties );
```

This class method returns the full ORBACUS ORB; use it in place of the **org.omg.CORBA.ORB.init** class method, and pass the String array and Properties arguments you would pass to the ORB initialization method.

If the application you are initializing requires a BOA, call the following method immediately after calling the **orbasec.SL2 init** class method:

```
public static org.omg.CORBA.BOA
BOA_init(
    String               argv[],
    java.util.Properties properties );
```

Note – **BOA_init** is an **orbasec.SL2** *class* method; it is not a member method of the **org.omg.CORBA.ORB** class. Indeed, you should be especially careful not to call the **BOA_init** member method on the ORB returned from **orbasec.SL2.init**; if you do, you will fail to fully initialize ORBASEC SL2.

ORBASEC SL2 provides two corresponding class initialization methods for initializing ORBASEC SL2 with Java applets.

```
public static org.omg.CORBA.ORB
init(
    java.applet.Applet  applet,
    java.util.Properties properties );
```

and

```
public static org.omg.CORBA.BOA
BOA_init(
    java.applet.Applet  applet,
    java.util.Properties properties );
```

These methods correspond roughly to the **org.omg.CORBA.ORB** and **org.omg.CORBA.BOA** initialization methods by the same names, and programmers should obey the same restrictions as those in using the **orbasec.SL2** standalone initialization methods. Programmers should also be aware that most Web browsers may place extra restrictions on Java applets, disallowing them from opening server sockets, and thus not allowing them to accept secure associations. Please consult your local Web browser documentation for enabling Java applets to open server sockets.

ORBASEC SL2 also provides “shortcut” class methods for initializing ORBASEC SL2 together with the ORB and the BOA. For standalone applications,

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

and for Java applets,

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

These methods merely call the **orbasec.SL2 init** and **BOA_init** class methods in sequence, passing the String array arguments (or Java applet) and Java Properties arguments to both.

To obtain the **org.omg.CORBA.ORB** and **org.omg.CORBA.BOA** initialized by any of these initialization methods, you may use the following accessors, available as class methods of **orbasec.SL2**:

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

Note that the return value from the BOA accessor may be null, if the BOA has not been initialized.

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 2.1.0 Final
```

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

```
prompt% java orbasec.SL2
ORBAsec SL2 2.1.0 Final
```

CHAPTER 4

Security Services

During initialization, ORBAsec SL2 adds a number of “initial” objects that make up the security services. These objects obtainable as “initial references” off the ORBAsec ORB. As of Security RTF 1.7, the nature and structure of these objects has changed considerably. In this chapter, we describe the new interfaces and how to obtain references to them off the ORB.

SecurityManager

Security RTF 1.7 introduces a new interface, **SecurityLevel2::SecurityManager**, which now has much of the functionality that used to be on the **SecurityLevel2::Current** object. The reasoning behind this modification is to split the functionality of ORB instance specific attributes and operations (SecurityManager) and thread-specific attributes and operations (Current).

The **SecurityLevel2::SecurityManager** object is a locality constrained CORBA object that maintains state information associated with the ORB instance, or other wise known as the “capsule”, or the process in which the ORB resides.

Getting the SecurityManager Object

The SecurityManager is obtainable as an initial reference off the ORB. To get a reference to it, use the ORB's **resolve_initial_references** method using the name "SecurityManager"

```
// Java
org.omg.CORBA.ORB orb = // The SL2 initialized ORB;
org.omg.SecurityLevel2.SecurityManager security_manager =
   org.omg.SecurityLevel2.SecurityManagerHelper.narrow(
           orb.resolve_initial_references("SecurityManager")
   );
```

Standard Attributes and Operations

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

principal_authenticator

This attribute's value is the **SecurityLevel2::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 87].

```
// IDL
readonly attribute PrincipalAuthenticator
                                     principal_authenticator;

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

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.

own_credentials

This attribute is the list of **SecurityLevel2::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.

```
// 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 **PrincipalAuthenticator** object. In fact, the only way a **Credentials** object makes it on the “own” credentials list, is by way of the **PrincipalAuthenticator** object.

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.

The **SecurityLevel2::Credentials** object is described in detail in [“Credentials” on page 126].

remove_own_credentials

This operation removes a given **SecurityLevel2::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.

The **SecurityLevel2::Credentials** object is described in detail in [“Credentials” on page 126].

get_target_credentials

This operation is for clients wishing to check the authentication of the target object reference. It returns a **SecurityLevel2::TargetCredentials** object containing the attributes of the target object reference.

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

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

The **SecurityLevel2::TargetCredentials** is described in detail in “Target Credentials” on page 141[].

This call performs a “nonexistent” call on the target object in order to cycle through any and all LOCATION_FORWARD calls that may happen below the in ORB’s object location layer. So, when this call returns the Credentials represent the principal that is holding on to the target object.

Note – One should be careful to set a RebindPolicy of NO_REBIND onto the object reference so that the connection to the target object's ORB is maintained and not automatically rebound. If this policy is not set on the object reference, he object can be redirected and rebound to a different principal throughout the use of the object reference to that target, and this would destroy any trust you had in checking the credentials.

---

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 this object, 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 this object, 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 this object, accessing this attribute raises a **CORBA::NO_IMPLEMENT** exception.

get_security_policy

This operation is used to retrieve security relevant policies associated with the capsule. This operation is used to retrieve policies about the scope of where the ORB instance lies, such as client side policies for determining which authentication mechanisms one should use, as this may be different for the different instances application residing in different domains. There is currently no standard way to an ORB instance with any of the policies, therefore ORBAsec SL2 does not support this operation.

This operation raises a **CORBA::NO_IMPLEMENT** exception.

ORBAsec SL2 Extensions to SecurityManager

ORBASEC SL2 makes several extensions to the **SecurityLevel2::SecurityManager** object. These extensions are in the form of an interface called **SecLev2::SecurityManager** that inherits from **SecurityLevel2::SecurityManager**.

The **SecLev2::SecurityManager** extensions allow the association of a set of Credentials objects with a specific Object, so that the proper credentials information gets placed in the object's IOR when it is exported to potential clients. Clearly, these operations only have meaning in the context of a server process which may be serving some collection of CORBA Objects.

Like the **SecurityLevel2::SecurityManager**, the **SecLev2::SecurityManager** object is returned from a call to the ORB's **resolve_initial_references** operation using the name "SecurityManager". This time, however, you should use the **orbasec.SecLev2.SecurityManagerHelper** class to narrow the reference to the appropriate type.

```
// Java
org.omg.CORBA.ORB orb = // The SL2 initialized ORB;
orbasec.SecLev2.SecurityManager security_manager =
   orbasec.SecLev2.SecurityManagerHelper.narrow(
           orb.resolve_initial_references(“SecurityManager”)
   );
```

The IDL definitions descriptions of the ORBASEC SL2 **SecLev2::SecurityManager** interface follows:

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

module SecLev2 {
  interface Current : SecurityLevel2::Current {

    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
    );
  };
};
```

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
);
```

Security Current

The **SecurityLevel2::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.

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”)
   );
```

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.

received_credentials

This read-only attribute is valid only in the context of an object (i.e., a “servant”) 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 client’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.

The **SecurityLevel2::ReceivedCredentials** interface is described in detail in “Received Credentials” on page 139.

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 an interface called **SecLev2::Current** that inherits from **SecurityLevel2::Current**.

The **SecLev2::Current** extensions allow the application programmer to restrict the set of Credentials on the current thread of execution that can be used to accept secure associations, so that the proper credentials information gets placed in the object’s IOR when it is exported to potential clients. Clearly, these operations only have meaning in the context of a server process which may be serving some collection of CORBA Objects.

Like the **SecurityLevel2::Current**, the **SecLev2::Current** object is returned from a call to the ORB's **resolve_initial_references** operation using the name "Security-Current". This time, however, you should use the **orbasec.SecLev2.CurrentHelper** class to narrow the reference to the appropriate type.

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

The IDL definitions and descriptions of the ORBASEC SL2 **SecLev2::Current** interface is below:

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

module SecLev2 {
  interface Current : SecurityLevel2::Current {

    attribute SecurityLevel2::CredentialsList
                                      accepting_credentials;
  };
};
```

.accepting_credentials

This 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.

PolicyCurrent

The **SecurityLevel2::PolicyCurrent** object is a locality constrained CORBA object that maintains state information associated with CORBA policies on the current execution context, such as in a multi-threaded execution model. This information is specific to the current thread of execution.

In previous versions of ORBASEC SL2, thread-specific policy related information was on the **SecurityLevel2::Current** interface. As of Security RTF 1.7, however, thread specific Policy information is moved off **Current** and placed on an entirely new interface, **SecurityLevel2::PolicyCurrent**. Moreover, the policy operations have been greatly simplified down to two operations, which we detail below.

Note – We must make one important caveat. Because the current version of ORBACUS (3.2) does not use the updated **CORBA::PolicyManager** interfaces, we have had to place the new interfaces in the **SecLev2** module. Adiron will update the module locations of these interfaces when the OMG has finalized the interfaces and ORBACUS is updated in accordance with them. Until then, ORBAsec SL2 developers should be mindful of the fact that the inheritance structure of the PolicyManager interface will likely change and should plan their development projects, accordingly.

Getting the PolicyCurrent Object

The **SecLev2::PolicyCurrent** object is returned from a call to the ORB's **resolve_initial_references** operation using the name "PolicyCurrent".

```
// Java
org.omg.CORBA.ORB orb = // The SL2 initialized ORB;
org.omg.SecLev2.PolicyCurrent policy_current =
   org.omg.SecLev2.PolicyCurrentHelper.narrow(
           orb.resolve_initial_references("PolicyCurrent")
   );
```

Standard Attributes and Operations

The attributes and options on the **SecLev2::PolicyCurrent** object are described below along with their values, semantics, and possible restrictions as pertaining to the ORBASEC SL2 implementation.

The following operations are the proposed (or about to be proposed) operations of the **CORBA::PolicyManager** interface, which the **CORBA::PolicyCurrent** interface inherits. 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_policy_overrides

This operation is thread specific and adds or sets the given policies for the current thread of execution, depending on the value of the **override_type** parameter.

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

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

The **CORBA::SetOverrideType** enumeration constant is defined in the CORBA module:

```
module CORBA {
    enum SetOverrideType {
        SET_OVERRIDE,
        ADD_OVERRIDE };
};
```

Using **SET_OVERRIDE** will replace any previously set policies with the supplied policies; using **ADD_OVERRIDES** will add the supplied policies, replacing any previously set policies if they are of the same 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_policy_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_policy_overrides(
       in CORBA::PolicyTypeSeq   policy_types
);

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

This operation returns an array of Policies, one for each type in the array of policy types. If the type array contains duplicate types, the return array of **Policy** objects will contain duplicate references. If a policy of a desired type is not found, a **CORBA::INV_POLICY** exception will be raised.

There is one exception to this rule: If a 0-length array of policy types is supplied, this operation will return *all* currently set policies for the current thread of execution.

CHAPTER 5

Principal Authenticator

Principal Authenticator

This section describes the application programmer’s use of the **SecurityLevel2::PrincipalAuthenticator** interface.

The PrincipalAuthenticator interface is implemented by a sole principal authenticator object. This object is a capsule specific object that resides on the **SecurityLevel2::SecurityManager** object. It is retrieved as follows:

```
// Java
org.omg.SecurityLevel2.SecurityManager mgr =
                   // .... get the security manager object
org.omg.SecurityLevel2.PrincipalAuthenticator pa =
                               mgr.principal_authenticator();
```

For details on the mechanism to get the **SecurityManager** object, see “Getting the SecurityManager Object” on page 70.

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 **PrincipalAu-**

thenticator object then places these operational credentials on the **SecurityManager** 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::SecurityName            security_name,
    in  any                               auth_data,
    in  Security::AttributeList           privileges,
    out Credentials                       creds,
    out any                               continuation_data,
    out any                               auth_specific_data
);

// Java
public org.omg.Security.AssocationStatus
```

```
authenticate(
    int                                     method,
    String                                  mechanism,
    String                                  security_name,
    org.omg.CORBA.Any                       auth_data,
    org.omg.Security.SecAttribute[]         privileges,
    org.omg.SecurityLevel2.CredentialsHolder
                                            creds,
    org.omg.CORBA.AnyHolder                 continuation_data,
    org.omg.CORBA.Any      Holder           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.

The method value must correspond with the data type that is held in the **org.omg.CORBA.Any** value passed to the **auth_data** parameter to the **PrincipalAuthenticator authenticate** method. The authentication method varies with each supported mechanism, and is described in detail in subsequent sections. For details about the authentication methods using Adiron's GSS-Kerberos security mechanism, see "Kerberos Authentication Methods and Data" on page 95. For details about the authentication methods using Adiron's SSL security mechanism, see "SSL Authentication Methods and Data" on page 109. For details about the authentication methods using Adiron's IIOP implementation, see "IIOP Authentication Methods and Data" on page 121.

Currently, there is no standard definition of authentication methods and corresponding data types; however, Adiron intends to propose a set of standard authentication methods for adoption to the CORBA Security Specification.

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::SecurityManager** object's **supported_mechanisms** attribute.

security_name

This parameter is a string 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 string.

auth_data

This parameter specifies the extra data needed to authenticate the principal. The type of this parameter is a **CORBA::Any**, and the type of the data held in the **Any** 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.

The CORBA object held in the **Any** should correspond directly with the authentication method tag supplied in the **method** parameter to this operation. ORBAsec SL2 provides several authentication methods and corresponding authentication data structures for authenticating principals. These authentication methods are specific to ORBAsec SL2 enabled security mechanisms. Adiron defines these authentication methods, their tags, and the corresponding data structures.They are detailed below.

Note – The ORBACUS implementation of the **CORBA::Any insert** and **extract** operations requires that the CORBA values being inserted or extracted be marshalled and unmarshalled, respectively. Therefore, the data structures that are inserted into the **auth_data** parameter must not contain null or invalid field values. Moreover, ORBASEC SL2 expects certain default values for fields that are not explicitly specified. Therefore, we have provided a utility class, **orbasec.corba.AuthUtil**, which should be used to create authentication structures with appropriate default values. The operations on this utility class are discussed below by way of example.

Some of the authentication methods in the ORBAsec SL2 suite of security mechanisms make use of the **SecLev2::CredentialsUsage** enumeration constant. This constant is used to specify how the acquired Credentials can be used in the application, and the definition of this constant mirrors that of the IETF's GSS-API. The range of values for this constant are

SecInitiateOnly	These Credentials may only be used to initiate a secure connection
SecAcceptOnly	These Credentials may only be used to accept (on the server side) a secure connection
SecInitiateAndAccept	These Credentials may be used to initiate and accept a secure connection

The value of this constant will affect the supported invocation and accepting options on the **Credentials** object created in the **authenticate** operation. See “invocation_options_supported” on page 133 and “accepting_options_supported” on page 130 for more information about the **Credentials** object’s invocation and accepting options.

For details about the authentication data using the GSS-Kerberos security mechanism, see “Kerberos Authentication Methods and Data” on page 95. For details about the authentication data using the SSL security mechanism, see “SSL Authentication Methods and Data” on page 109. For details about the authentication data using the IIOP implementation, see “IIOP Authentication Methods and Data” on page 121.

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 **SecurityManager** object and places the new credentials in the manager’s own credentials list should the return value from authenticate be **SecAuthSuccess**. If the **authenticate** operation returns **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 **SecAuth-Success** and places the fully enabled **Credentials** object on the **SecurityManager** 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 **SecAuth-Success** and places the fully enabled **Credentials** object on the **SecurityManager** 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   org.omg.CORBA.Any     response_data,
    in   Credentials           creds,
    out  org.omg.CORBA.Any     continuation_data,
    out  org.omg.CORBA.Any     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 for these mechanisms.

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 **SecurityManager** object'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 **SecAuth-**

Continue, 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. 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
);

// Java
public int[] get_supported_authen_methods( String mechanism );
```

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 is a **java.lang.String** representing a Kerberos name. Typically, you will want to supply the full Kerberos principal name, including the Kerberos realm, such as in “bart@MYREALM.COM”. If the **security_name** parameter is an empty string (““), then the name stored in the specified or default Kerberos credentials cache will be used. See the following section for how to specify a Kerberos credentials cache.

Kerberos Authentication Methods and Data

The authentication data is the value of the **auth_data** parameter for the **authenticate** operation. The type of this parameter is **CORBA::Any**, meaning that it can carry any CORBA type. In the case of authentication using Adiron’s GSS-Kerberos security mechanisms, you must specify one of Adiron’s supported authentication methods in the **method** parameter to the **authenticate** operation and supply a **CORBA::Any** in the **auth_data** parameter. The **Any**, when extracted, should contain a fully defined data structure which corresponds to the authentication method specified. If the authentication method and unmarshalled authentication structure do not match, a **CORBA::BAD_PARAM** exception will be raised.

Adiron provides 3 authentication methods for Kerberos authentication: one for authentication using a Kerberos password, one for authenticating servers using a Kerberos keytab file, and one for authenticating using previously establish Kerberos session credentials. The descriptions of these authentication methods are given below.

Note – Previous versions of ORBAsec SL2 used Java properties to define authentication method parameters to the **PrincipalAuthenticator authenticate** operation. However, due to its cumbersome logic, Adiron no longer supports this authentication method with its security mechanisms.

KerberosPassword Authentication Method

The KerberosPassword authentication method is indicated by use of the **SecLev2::SecKerberosPassword** value in the **method** parameter to the **authenticate** operation.

Use this authentication method to establish Credentials using a Kerberos password. The password should match the Kerberos principal identified in the **security_name** parameter to the **PrincipalAuthenticator authenticate** operation.

Note – Using this authentication method will cause the Kerberos library to contact the specified Kerberos KDC to acquire a ticket-granting ticket to connect to kerberized servers.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::KerberosPassword** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct KerberosPassword {
        string           config;
        string           password;
        string           lifetime;
        boolean          proxiable;
        boolean          forwardable;
        string           renewablelife;
        CredentialsUsage usage;
};
```

You should use the **AuthUtil**.**default_KerberosPassword** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

config

This field contains the name of the Kerberos configuration file. This file contains information pertaining to the configuration of the kerberos configuration and is specific to the MIT Kerberos implementation. Such information includes the network location of the KDC, and other parameters. Consult your local Kerberos documentation for details about the specific format of this file for your platform.

The config specification should have a URL format:

type:*location*

Currently, the only valid *type* is **FILE:**. Use **location** to specify the path (absolute or relative), in a format local to your platform, of the Kerberos configuration file you would like to use.

If the **config** field contains an empty string (““), the default configuration file for the Kerberos installation is used. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosPassword** method is ““.

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 field must contain the password in string form.

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. To specify the system default for your Kerberos installation, supply an empty value (““). The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosPassword** method is ““.

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. Please see the Internet RFC 1510[2] for an explanation of the details. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosPassword** method is **false**.

forwardable

This 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. Please see the Internet RFC 1510[2] for an explanation of the details. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosPassword** method is **false**.

renewablelife

This field specifies the amount of time the Kerberos credentials can be renewed. 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. To specify the system default for your Kerberos installation, supply an empty value (""). The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosPassword** method is "".

usage

This field specifies how the acquired Credentials may be used. The value of this field will affect the supported invocation and accepting options on the **Credentials** object created in the **authenticate** operation. See "invocation_options_supported" on page 133 and "accepting_options_supported" on page 130 for more information about the **Credentials** object's invocation and accepting options. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosPassword** method is **SecInitiateOnly**.

Note – If the value of this field is **SecAcceptOnly** or **SecInitiateAndAccept**, you must initialize the ORBASEC SL2 SECIOP security mechanism in server mode. Failure to do so will result in the **CORBA::BAD_PARAM** exception. See "orbasec.seciop" on page 55 for information about initializing the SECIOP mechanism in server mode.

The following example illustrates how to use the KerberosPassword authentication method to create an authentication data structure, using a local configuration file and hard-wired password.

```
orbasec.SecLev2.KerberosPassword data =
       orbasec.corba.AuthUtil.default_KerberosPassword();
data.config   = "FILE:./orbasec_krb5.conf";
data.password = "mypassword";
data.usage    =
       orbasec.SecLev2.CredentialsUsage.SecInitiateAndAccept;
org.omg.CORBA.ORB orb = // obtain reference to ORB
int method = orbasec.SecLev2.SecKerberosPassword.value;
org.omg.CORBA.Any auth_data = orb.create_any();
orbasec.SecLev2.KerberosPasswordHelper.insert(
       auth_data, data );
```

KerberosService Authentication Method

The KerberosService authentication method is indicated by use of the **SecLev2::SecKerberosService** value in the **method** parameter to the **authenticate** operation.

Use this authentication method to establish Credentials for a Kerberos service using a Kerberos *keytab* file.

Note – **Credentials** established with this authentication method may *only* be used to accept secure associations; they may not be used to make secure invocations. You must initialize the ORBAsec SL2 SECIOP security mechanism in server mode if you are going to use this authentication method. See “orbasec.seciop” on page 55 for information about initializing the SECIOP mechanism in server mode.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::KerberosService** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct KerberosService {
        string          config;
        string          keytab;
};
```

You should use the **AuthUtil default_KerberosService** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

config

This field contains the name of the Kerberos configuration file. This file contains information pertaining to the configuration of the kerberos configuration and is specific to the MIT Kerberos implementation. Such information includes the network location of the KDC, and other parameters. Consult your local Kerberos documentation for details about the specific format of this file for your platform.

The config specification should have a URL format:

type`:`*location*

Currently, the only valid *type* is **FILE:**. Use **location** to specify the path (absolute or relative), in a format local to your platform, of the Kerberos configuration file you would like to use.

If the **config** field contains an empty string (““), the default configuration file for the Kerberos installation is used. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosService** method is ““.

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.

keytab

This field is used to specify the Kerberos *keytab* file, a feature specific to MIT Kerberos. This file contains keys for Kerberos “service” principals, and is generally created and distributed by the Kerberos administrator, using a program such as **kadmin**. This file must contain the secret key for the principal identified in the **security_name** parameter to the **authenticate** method.

Note – Since the *keytab* file contains the secret keys for Kerberos principals, care should be taken to make this file readable only by privileged users, such as `root`. However, the user running ORBASEC SL2 must have read permission on this file if the KerberosService authentication method is used.

The *keytab* specification should have a URL format:

type`:`*location*

Currently, the only valid *type* is **FILE:**. Use **location** to specify the path (absolute or relative), in a format local to your platform, of the Kerberos *keytab* file you would like to use.

If the **keytab** field contains an empty string (““), the default *keytab* file for the Kerberos installation is used. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosService** method is ““.

On Unix systems, the default *keytab* file 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 – 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.

The following example illustrates how to use the KerberosService authentication method to create an authentication data structure, using a local configuration file and *keytab* file.

```
orbasec.SecLev2.KerberosService data =
       orbasec.corba.AuthUtil.default_KerberosService();
data.config   = "FILE:./orbasec_krb5.conf";
data.keytab   = "FILE:./orbasec_krb5.keytab";
org.omg.CORBA.ORB orb = // obtain reference to ORB
int method = orbasec.SecLev2.SecKerberosService.value;
org.omg.CORBA.Any auth_data = orb.create_any();
orbasec.SecLev2.KerberosServiceHelper.insert(
       auth_data, data );
```

KerberosSession Authentication Method

The KerberosSession authentication method is indicated by use of the **SecLev2::SecKerberosSession** value in the **method** parameter to the **authenticate** operation.

Use this authentication method to establish Credentials using previously established Kerberos credentials, such as via a program like **kinit**.

Note – Credentials established using this authentication method may *only* be used to make secure invocations; they may not be used to accept secure associations.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::KerberosSession** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct KerberosSession {
        string          config;
        string          cache_name;
};
```

You should use the **AuthUtil default_KerberosSession** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

config

This field contains the name of the Kerberos configuration file. This file contains information pertaining to the configuration of the kerberos configuration and is specific to the MIT Kerberos implementation. Such information includes the network location of the KDC, and other parameters. Consult your local Kerberos documentation for details about the specific format of this file for your platform.

The config specification should have a URL format:

type`:`*location*

Currently, the only valid *type* is **FILE:**. Use **location** to specify the path (absolute or relative), in a format local to your platform, of the Kerberos configuration file you would like to use.

If the **config** field contains an empty string (““), the default configuration file for the Kerberos installation is used. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosSession** method is ““.

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.

cache_name

This field names the location of Kerberos credentials cache that you want to use. This cache is specific to the MIT Kerberos implementation and is used to store transient data about a Kerberos principal’s credentials.

The credentials cache specification should have a URL format:

type`:`*location*

Currently, the only valid *type* is **FILE:**. Use **location** to specify the path (absolute or relative), in a format local to your platform, of the Kerberos credentials cache file you would like to use.

If the **cache_name** field contains an empty string (““), the default credentials cache file for the Kerberos installation is used. The default value for this field in the authentication structure returned from the **AuthUtil default_KerberosSession** method is ““.

On most Unix systems, the default credentials cache file is “`/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.

Note – If the **security_name** parameter to the **authenticate** operation is nonempty, 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.

The following example illustrates how to use the KerberosSession authentication method to create an authentication data structure, using a local configuration file and credentials cache file.

```
orbasec.SecLev2.KerberosSession data =
       orbasec.corba.AuthUtil.default_KerberosSession();
data.config   = "FILE:./orbasec_krb5.conf";
data.keytab   = "FILE:./orbasec_krb5.ccache";
org.omg.CORBA.ORB orb = // obtain reference to ORB
int method = orbasec.SecLev2.SecKerberosSession.value;
org.omg.CORBA.Any auth_data = orb.create_any();
orbasec.SecLev2.KerberosSessionHelper.insert(
       auth_data, data );
```

KerberosServiceClient Authentication Method

The KerberosServiceClient authentication method is indicated by use of the **SecLev2::SecKerberosServiceClient** value in the **method** parameter to the **authenticate** operation.

Use this authentication method to establish service credentials from a Kerberos keytab file that also has the capability to be a client. This method is a combination of the use of the KerberosService method, which names the keytab file, and the KerberosPassword method which names client characteristics of the invocation credentials. No password is needed as the service key is contained in the keytab file.

Note – Using this authentication method will cause the Kerberos library to contact the specified Kerberos KDC to acquire a ticket-granting ticket to connect to kerberized servers.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::KerberosServiceClient** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct KerberosServiceClient {
        string           config;
        string           keytab;
        string           lifetime;
        boolean          proxiable;
        boolean          forwardable;
        string           renewablelife;
        CredentialsUsage usage;
};
```

You should use the **AuthUtil.default_KerberosServiceClient** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

You will find the descriptions of the fields of this structure in the above sections for KerberosPassword and KerberosService authentication methods.

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 may generally be left empty (““). In this case, the principal’s Subject Directory Name (SubjectDN) is retrieved from the public key of the first certificate in the certificate chain that is specified in the authentication data. If, on the other hand, the value of the **security_name** parameter is nonempty, it is compared with the SubjectDN in the certificate. If they do not match, a **CORBA::BAD_PARAM** exception is raised.

In the absence of any standards for testing Directory Names for equality, the general practice should be to leave the **security_name** parameter empty and let the principal’s SubjectDN come from the certificate.

SSL Authentication Methods and Data

The authentication data is the value of the **auth_data** parameter for the **authenticate** operation. The type of this parameter is **CORBA::Any**, meaning that it can carry any CORBA type. In the case of authentication using Adiron’s SSL security mechanism, you must specify one of Adiron’s supported authentication methods in the **method** parameter to the **authenticate** operation and supply a **CORBA::Any** in the **auth_data** parameter. The **Any**, when extracted, should contain a fully defined data structure which corresponds to the authentication method specified. If the authentication method and unmarshalled authentication structure do not match, a **CORBA::BAD_PARAM** exception will be raised.

Adiron provides the following authentication methods for use with the SSL mechanism:

- **SSLKeyStoreWithStorePass** - using a Java KeyStore which is integrity protected with a password,
- **SSLKeyStoreNoStorePass** - using a Java KeyStore which is not integrity protected,
- **SSLEncryptedPrivateKeyAndCertificates** - using an encrypted private key and certificate chain which have been encoded by the user,

- **SSLPrivateKeyAndCertificates** - using an unencrypted private key and certificate chain which have been encoded by the user, and
- **SSLAnonymous** - using anonymous SSL credentials.

The descriptions of these authentication methods are given below.

Note – Previous versions of ORBASEC SL2 used Java properties to specify the location of private key and certificate files. Unfortunately, the interfaces for manipulating these files were vendor-specific and not widely used by ORBASEC SL2 customers. We have moved to the more broadly adoptable Java KeyStore architecture for storing private keys and certificates. Adiron no longer supports the older private key and certificate file format used in previous versions of ORBASEC SL2. We do, however, provide a tool for adding the keys and certificates in the older key and certificate files to a Java KeyStore file. See “Other Java Utility Classes” on page 209 for more information.

SSLKeyStoreWithStorePass Authentication Method

The SSLKeyStoreWithStorePass authentication method is indicated by use of the **SecLev2::SecSSLKeyStoreWithStorePass** value in the **method** parameter to the **authenticate** operation.

Use this authentication method to establish Credentials using a Java KeyStore file or resource which is integrity protected with a pass phrase.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::SSLKeyStoreWithStorePass** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct SSLKeyStoreWithStorePass {
        string           keystore;
        string           storetype;
        string           storepass;
        string           alias;
        string           keypass;
        sequence<string> certificate_aliases;
        CredentialsUsage usage;
        unsigned short   ssl_port
};
```

You should use the **AuthUtil default_SSLKeyStoreWithStorePass** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

keystore

This field designates the Java Keystore file or resource from which the principal's private key and certificates are obtained.

This specification should have a URL format:

type`:`*location*

where *type* is `FILE`, `HTTP`, `FTP`, etc., and *location* designates the corresponding path to the resource. You may use a place-name relative to the current working directory if you use the `FILE` designator.

A non-empty value must be supplied for this field.

storetype

This field specifies the store type, which varies according to the Java security provider used to create the Java keystore.

Note – Under the JDK-1.2, you must install a security provider in your Java installation and specify a default KeyStore type. Consult your local JDK-1.2 documentation for instructions about installing security providers.

An empty value ("") for this field instructs ORBAsec SL2 to use the default Java KeyStore provider for the current Java installation. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLKeyStoreWithStorePass** method is "".

storepass

This field specifies the pass phrase to use to automatically perform an integrity check on the specified KeyStore. A non-empty value must be specified for this field.

If you would like to use a KeyStore that is not integrity protected with a passphrase, use the **SecLev2::SecSSLKeyStoreNoStorePass** authentication method. Otherwise, an exception will be raised. See "SSLKeyStoreNoStorePass Authentication Method" on page 113 for more information.

alias

This field specifies the alias in the Java KeyStore which is used to select the principal's private key and certificate chain. A non-empty value must be specified for this field.

Note – Be sure that the alias specified designates a "key" entry, not a "trusted certificate" entry, in the specified Java KeyStore. If the alias does not designate a private key in the specified KeyStore, a **CORBA::BAD_PARAM** exception will be raised.

keypass

This field specifies the passphrase used to unlock the private key selected using the **alias** field. A non-empty value must be specified for this field.

certificate_aliases

This field is a comma-separated list of aliases that can be used to select certificate entries in the Java KeyStore. These certificates are used to specify which authorities are trusted as signers for digital certificates for the peer authentication process. For more information about how trusted certificates are used in the ORBAsec SL2 SSLIOP module, see "The SSLIOP Authentication Model" on page 122.

An empty value ("") for this field instructs ORBAsec SL2 to use the *second* certificate in the principal's certificate chain. This certificate is the authority who signed the principal's certificate, and is taken to be the default certificate authority, in the absence of any explicitly specified trusted certificates through this field. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLKeyStoreWithStorePass** method is "".

usage

This field specifies how the acquired Credentials may be used. The value of this field will affect the supported invocation and accepting options on the **Credentials** object created in the **authenticate** operation. See "invocation_options_supported" on page 133 and “accepting_options_supported” on page 130 for more information about the **Credentials** object’s invocation and accepting options. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLKeyStoreWithStorePass** method is **SecInitiateOnly**.

Note – If the value of this field is **SecAcceptOnly** or **SecInitiateAndAccept**, you must initialize the ORBASEC SL2 SSLIOP security mechanism in **server** mode. Failure to do so will result in the **CORBA::BAD_PARAM** exception. See “orbasec.ssliop” on page 56 for information about initializing the SSLIOP mechanism in server mode.

---

ssl_port

This field specifies a requested port number to use when establishing an SSL connection. A value of 0 instructs ORB to automatically select a port. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLKeyStoreWithStorePass** method is 0.

Note – Some operating systems require privileged access in order to open a port within a specified range. Consult your local operating system documentation for details about what ports you, as a user, may open.

The following example illustrates how to use the **SSLKeyStoreWithStorePass** authentication method to create an authentication data structure.

```
orbasec.SecLev2.SSLKeyStoreWithStorePass data =
  orbasec.corba.AuthUtil.default_SSLKeyStoreWithStorePass();
data.keystore     = "FILE:./demo.keystore";
data.storetype    = "IAIKKeyStore";
data.storepass    = “I can see two tiny pictures of myself“;
data.alias        = “wsb“;
data.keypass      = “and there’s one in each of your eyes“;
data.usage        =
       orbasec.SecLev2.CredentialsUsage.SecInitiateAndAccept;
org.omg.CORBA.ORB orb = // obtain reference to ORB
int method = orbasec.SecLev2.SeSSLKeyStoreWithStorePass.value;
org.omg.CORBA.Any auth_data = orb.create_any();
orbasec.SecLev2.SSLKeyStoreWithStorePassHelper.insert(
       auth_data, data );
```

SSLKeyStoreNoStorePass Authentication Method

The SSLKeyStoreNoStorePass authentication method is indicated by use of the **SecLev2::SecSSLKeyStoreNoStorePass** value in the **method** parameter to the **authenticate** operation.

Use this authentication method to establish Credentials using a Java KeyStore file or resource in which integrity of the keystore is *not* an issue.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::SSLKeyStoreNoStorePass** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct SSLKeyStoreNoStorePass {
        string           keystore;
        string           storetype;
        string           alias;
        string           keypass;
        sequence<string> certificate_aliases;
        CredentialsUsage usage;
        unsigned short   ssl_port
};
```

You should use the **AuthUtil default_SSLKeyStoreNoStorePass** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

The fields of this structure are the same as they are for the SSLKeyStoreWithStorePass structure, with the exception that there is no **storepass** field in this structure. For details about the rest of the fields of this structure, see “SSLKeyStoreWithStorePass Authentication Method” on page 110.

SSLPrivateKeyAndCertificates Authentication Method

The **SecLev2::SecSSLPrivateKeyAndCertificates** authentication method is used to specify an encoded private key and certificate chain, and, optionally, a list of certificates for trusted authorities.

Use this authentication method if you have obtained your private key and certificate chain from some external resource (e.g., a file, or some external API) and wish to use the private key and certificates to authenticate SSL credentials.

Both the private key and certificate chain (and, optionally, the list of trusted certificates) must be encoded in a format that ORBAsec SL2 can understand. Since the ORBAsec SL2 SSLIOP module uses the Java Cryptography Architecture (JCA) for managing private keys and certificates, this authentication method understands key formats and algorithms that the JCA and underlying Java security provider supports. For details about key format and algorithm names, we refer the reader to Appendix A of [3].

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::SSLPrivateKeyAndCertificates** structure. The IDL definition and fields of this structure are as follows:

```
struct SSLPrivateKeyAndCertificates {
    Security::Opaque  encoded_private_key;
    string            key_algorithm;
    string            key_format;
    string            security_provider;
    Security::Opaque  encoded_certificate_chain;
    Security::Opaque  encoded_trusted_authorities;
    CredentialsUsage  usage;
    unsigned short    ssl_port;
};
```

You should use the **AuthUtil default_SSLPrivateKeyAndCertificates** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

encoded_private_key

This field contains the **Security::Opaque** (**byte[]** under the Java mapping) encoding of the private key for the principal being authenticated.

The private key, when decoded, should not be an encrypted private key. If you only have access to an encrypted private key, use the SecLev2::SSLEncryptedPrivateKeyAndCertificates authentication method. See “SSLEncryptedPrivateKeyAndCertificates Authentication Method” on page 118 for more information.

Note – The private key must correspond with the first public key in the supplied certificate chain, passed through the **encoded_certificate_chain** field (see below). If it does not, then a **CORBA::NO_PERMISSION** exception will be raised when ORBAsec SL2 attempts to verify the certificate chain.

This field must contain a non-empty byte array, or else a **CORBA::BAD_PARAM** exception will be raised during authentication.

key_algorithm

This field specifies the key algorithm used in the encoded private key. Use one of the standard algorithm identifiers specified in Appendix A of [3]. A non-empty value must be specified for this field.

key_format

This field specifies the key format of the encoded private key. Use one of the standard format identifiers specified in Appendix A of [3]. A non-empty value must be specified for this field.

security_provider

This field specifies the Java security provider you would like to use to decode the encoded private key. You may need to designate a specific security provider if, for example, the default security provider for your Java installation does not support a particular key algorithm or encoding.

Note – Under the JDK-1.2, you must install a security provider in your Java installation and specify a default provider. Consult your local JDK-1.2 documentation for instructions about installing security providers.

An empty value (““) for this field instructs ORBASEC SL2 to use the default security provider for the current Java installation. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLPrivateKayAndCertificates** method is ““.

encoded_certificate_chain

This field contains an encoding of an X.509 certificate chain, ostensibly the certificate chain corresponding with the principal’s private key specified in the

encoded_private_key field. The public key in the first certificate in this chain should be the public key corresponding with this private key.

Note – ORBAsec SL2 only supports X.509 certificates and certificate chains, and only one encoding, ASN.1, for these certificates. If your organization has a need for other certificate types and encodings, please contact Adiron.

encoded_trusted_authorities

This field contains an encoding of a sequence of X.509 certificates. These certificates are used to specify which authorities are trusted as signers for digital certificates exchanged between SSL Peers. For more information about how trusted certificates are used in the ORBAsec SL2 SSLIOP module, see “The SSLIOP Authentication Model” on page 122.

An empty value (““) for this field instructs ORBAsec SL2 to use the *second* certificate in the principal’s certificate chain. This certificate is the authority who signed the principal’s certificate, and is taken to be the default certificate authority, in the absence of any explicitly specified trusted certificates through this field.

Note – ORBAsec SL2 only supports X.509 certificates and certificate chains, and only one encoding, ASN.1, for these certificates. If your organization has a need for other certificate types and encodings, please contact Adiron.

usage

This field specifies how the acquired Credentials may be used. The value of this field will affect the supported invocation and accepting options on the **Credentials** object created in the **authenticate** operation. See “invocation_options_supported” on page 133 and “accepting_options_supported” on page 130 for more information about the **Credentials** object’s invocation and accepting options. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLPrivateKeyAndCertificates** method is **SecInitiateOnly**.

Note – If the value of this field is **SecAcceptOnly** or **SecInitiateAndAccept**, you must initialize the ORBASEC SL2 SSLIOP security mechanism in server mode. Failure to do so will result in the **CORBA::BAD_PARAM** exception. See “orbasec.ssliop” on page 56 for information about initializing the SSLIOP mechanism in server mode.

ssl_port

This field specifies a requested port number to use when establishing an SSL connection. A value of **0** instructs ORB to automatically select a port. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLPrivateKeyAndCertificates** method is **0**.

Note – Some operating systems require privileged access in order to open a port within a specified range. Consult your local operating system documentation for details about what ports you, as a user, may open.

SSLEncryptedPrivateKeyAndCertificates Authentication Method

The **SecLev2::SecSSLEncryptedPrivateKeyAndCertificates** authentication method is used to specify an encoded and encrypted private key and certificate chain, and, optionally, a list of certificates for trusted authorities.

Use this authentication method if you have obtained your private key and certificate chain from some external resource (e.g., a file, or some external API), where the private key is encrypted and wish to use the private key and certificates to authenticate SSL credentials.

Note – There is no danger of exposing a private key if it is not encrypted, since the key is only used locally to acquire credentials; you need only use this authentication method if you do not have the resources or inclination to decrypt a private key you may have obtained via some external resource.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::SSLEncryptedPrivateKeyAndCertificates** structure. The IDL definition and fields of this structure are as follows:

```
struct SSLEncryptedPrivateKeyAndCertificates {
    Security::Opaque  encoded_private_key;
    string            key_algorithm;
    string            key_format;
    string            key_pass,
    string            security_provider;
    Security::Opaque  encoded_certificate_chain;
    Security::Opaque  encoded_trusted_authorities;
    CredentialsUsage  usage;
    unsigned short    ssl_port;
};
```

You should use the **AuthUtil default_SSLEncryptedPrivateKeyAndCertificates** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

The fields of the **SecLev2::SSLEncryptedPrivateKayAndCertificates** structure are identical to the fields of the **SecLev2::SSLPrivateKayAndCertificates** structure, with the exception of the **key_pass** field, described below. For details about the rest of the fields of this structure, see “SSLPrivateKeyAndCertificates Authentication Method” on page 114.

key_pass

This field contains the passphrase used to unlock the private key stored in the **encoded_private_key** field. A non-empty value must be supplied for this field.

SSLAnonymous Authentication Method

The **SecLev2::SecSSLAnonymous** authentication method is used to acquire anonymous SSL Credentials. The SSLIOP mechanism will select an anonymous CipherSuite, depending on the advertised capabilities of the SSL Peer. The SSL peer will not be able to establish trust in your identity if you use this authentication method.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::SSLAnonymous** structure. The IDL definition and fields of this structure are as follows:

```
struct SSLAnonymous {
    CredentialsUsage  usage;
    unsigned short    ssl_port;
};
```

You should use the **AuthUtil default_SSLAnonymous** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

usage

This field specifies how the acquired Credentials may be used. The value of this field will affect the supported invocation and accepting options on the **Credentials** object created in the **authenticate** operation. See “invocation_options_supported” on page 133 and “accepting_options_supported” on page 130 for more information about the **Credentials** object’s invocation and accepting options. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLAnonymous** method is **SecInitiateOnly**.

Note – If the value of this field is **SecAcceptOnly** or **SecInitiateAndAccept**, you must initialize the ORBASEC SL2 SSLIOP security mechanism in server mode. Failure to do so will result in the **CORBA::BAD_PARAM** exception. See “orbasec.ssliop” on page 56 for information about initializing the SSLIOP mechanism in server mode.

ssl_port

This field specifies a requested port number to use when establishing an SSL connection. A value of **0** instructs ORB to automatically select a port. The default value for this field in the authentication structure returned from the **AuthUtil default_SSLAnonymous** method is **0**.

Note – Some operating systems require privileged access in order to open a port within a specified range. Consult your local operating system documentation for details about what ports you, as a user, may open.Authentication of IIOP Credentials

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 manner.

Mechanism

The mechanism must be specified as “IIOP”.

Security Name

The security name must be empty (““).

IIOP Authentication Methods and Data

ORBASEC SL2 provides only 1 authentication method for IIOP authentication, **SecLev2::SecIIOP**.

IIOP Authentication Method

The **SecLev2::SecIIOP** authentication method is used to acquire IIOP Credentials. Connections made with these credentials will not be encrypted and will provide no data integrity; nor will you be able to establish trust in a client or server using these credentials to make a “secure” association.

The **CORBA::Any** passed to the **auth_data** parameter should carry a **SecLev2::IIOP** structure. The IDL definition and fields of this structure are as follows:

```
// IDL
struct IIOP {
    CredentialsUsage  usage;
    unsigned short    ssl_port;
};
```

You should use the **AuthUtil default_IIOP** method to create an instance of this structure with marshallable fields and defaults values, where appropriate. You may then fill the returned data structure with suitable values.

usage

This field specifies how the acquired Credentials may be used. The value of this field will affect the supported invocation and accepting options on the **Credentials** object created in the **authenticate** operation. See “invocation_options_supported” on page 133 and “accepting_options_supported” on page 130 for more information about the **Credentials** object’s invocation and accepting options. The default value for this field in the authentication structure returned from the **AuthUtil default_IIOP** method is **SecInitiateOnly**.

Note – If the value of this field is **SecAcceptOnly** or **SecInitiateAndAccept**, you must initialize the ORBASEC SL2 IIOP security mechanism in server mode. Failure to do so will result in the **CORBA::BAD_PARAM** exception. See “orbasec.iiop” on page 57 for information about initializing the IIOP mechanism in server mode.

iiop_port

This field specifies a requested port number to use when establishing an IIOP connection. A value of **0** instructs ORB to automatically select a port. The default value for this field in the authentication structure returned from the **AuthUtil default_IIOP** method is **0**.

The SSLIOP Authentication Model

Public key systems, by their very nature, raises issues with verification of authentication information. Unlike the Kerberos module, where authentication verification is performed by contacting a trusted third party to verify the identity of a remote principal, the SSLIOP module uses public key X.509 certificate chains for establishment of authentication, where the peer delivers its authentication information. Each certificate in a certificate chain contains the public key for a principal, which is signed by the principal who’s public key is contained in the next certificate in the certificate chain. Typically (though not necessarily), the last certificate in a certificate chain is self signed, i.e. a *root* certificate. A root certificate’s public key is signed using the private key of the same principal to whom the public key belongs. However, anyone can create a self-signed certificate, and hence a certificate chain with a “bogus” *root* certificate. How, then, does one trust a *root* certificate?

Most public key systems use the notion of a Certificate Authority (CA) to address this problem. A CA is an entity (such as the Post Office, or a notary) that issues certificates, and these certificates are distributed with programs, such as a Web browser, and are used when it comes time to verify the identity of another party.

The ORBASEC SL2 SSLIOP module uses this model, as well, except that the certificates for the verification authorities must be provided during authentication, i.e. the trusted third party.

If you are using one of the authentication method for establishing SSL Credentials (See "KerberosServiceClient Authentication Method" on page 105) you will notice that you have the option to specify a collection of trusted certificates. If you specify such a collection, the certificates you designate will be the certificates you trust as signers of certificates in using these Credentials to make or service object invocations of an SSL peer.

When an SSL connection is made the ORBASEC ORB will receive the SSL peer's certificate chain. It will then verify this certificate chain, ensuring first that the list of certificates forms a valid chain, and secondly that *last* certificate in the certificate chain is the one of certificates of the trusted authorities specified during authentication. If the last certificate in the chain is not among the trusted authorities, then the SSL handshake fails, and a **CORBA::NO_PERMISSION** exception is raised.

If no authorities are specified during authentication, then ORBASEC SL2 takes the second certificate in the authenticating principal's own certificate chain to be the sole trusted authority. This default behavior, while it may seem restrictive, is commonplace; typically, an organization or organizational unit may have a single trusted authority which all members of the organization use. So even the default behavior is serviceable for many, if not the majority, of applications.

Note – There is no way to set the trusted authorities for a specific capsule or thread. This is largely because the requirement for and format of trusted authorities is highly dependent on the SSL security mechanism. Instead, trusted authorities are associated with SSL **Credentials** objects, since it is a user's credentials are used to make secure associations.

Note – The mechanism described here is an authentication verification mechanism and does not constitute establishment of trust for the verified identity! An application should examine the identity and make it's own trust determination of the principal, only knowing that ORBAsec SL2 has verified the identity of the principal to the extent of the information supplied by the application.

CHAPTER 6

Credentials

What are Credentials?

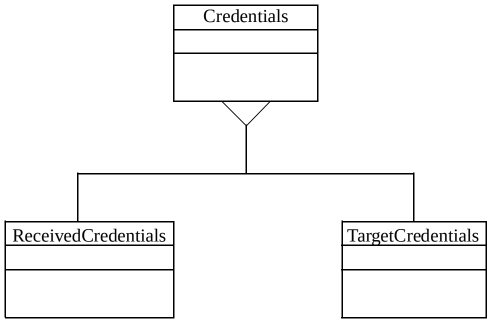
Credentials are the application programmer's interface to querying of security related attributes belonging to the application itself and of any clients making invocations. Also, one may examine the Credentials of a server. Credentials come in three flavors, "own" credentials, "received", and "target" 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. The target 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 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  any                           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 163.

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();
```

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

TABLE 5. **SSL Cipher Suite Accepting 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 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
MechUtil.SSL_DHE_DSS_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation
MechUtil.SSL_DHE_RSA_MECH	Integrity, Confidentiality, DetectReplay, DetectMisordering, EstablishTrustInClient, EstablishTrustInTarget,NoDelegation

TABLE 6. SSL Cipher Suite Invocation Options Supported

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

TABLE 6. 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, 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.

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_feature(
    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. Not 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 this parameter 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 the 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 “Security Attributes of Credentials” on page 142 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  any  refresh_data
);

// Java
public boolean refresh( org.omg.CORBA.Any 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** argument is ignored.

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 their 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 AttributeType 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 7. **CORBA Family Definer**

CORBA Family Definer
0

CORBA currently defines two families of attributes.

TABLE 8. **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.

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 family 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 9. 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 10. Adiron Families

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

Adiron also defines the following attribute types.

TABLE 11. 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

As of Security RTF 1.7, the defining authority is an octet sequence taken to be an ASN.1 encoding of an OID. This OID represents and entity which defines the encoding of the **value** field of a **SecAttribute**. Any empty value (0-length byte array) for the defining authority is specified to mean that the defining authority is the OMG, and, in addition, that the value field is a UTF-8 encoding of a string value.

All security attributes created by ORBASEC SL2 use the OMG as the defining authority for the encoding of the value field. That is, the defining authority is a 0-length byte array, and the value field is UTF-8 encoding of a string value.

Value

The values of the Adiron security attributes are UTF-8 encodings of string values.

TABLE 12. **Adiron Attribute Values**

Attribute Type	**Value Description**
Security Mechanism Type	This value is an UTF-8 encoding of a security mechanism. The string that is encoded is the mechanism type of the credentials.
Local Host Address	This value is an UTF-8 encoding of an IP address in canonical string form. This IP address is that of the local machine.
Local Port Number	This value is an UTF-8 encoding of an integer representing the port number of the connection on the local machine.
Peer Host Address	This value is an UTF-8 encoding of an IP address in canonical string form. This IP address is that of the remote host. This attribute only exists in ReceivedCredentials or TargetCredentials.
Peer Port Number	This value is an UTF-8 encoding of an integer representing the port number of the connection on the remote machine. This attribute only exists in ReceivedCredentials or TargetCredentials.

TABLE 12. **Adiron Attribute Values**

Attribute Type	**Value Description**
Subject Identity	This value is an UTF-8 encoding of a value specific to the mechanism (see below). It is the identify attribute of the principal.
Issuer Identity	This value is an UTF-8 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 UTF-8 encoding of a KerberosName that is the principal's name, such as "bart@MYREALM.COM".

If you have the ORBASEC SL2-SSL distribution, the **value** field of the Subject Identity and the Issuer Identity attributes contain the UTF-8 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. 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 UTF-8 encoding of the string "anonymous".

CORBA Family 1 Public

The **Public** attribute is defined by CORBA Family 1, and its attribute type identifier is 1. This attribute has only one value, which is a UTF-8 encoding of an empty string. It is used merely to indicate that any fully initialized Credentials object represents a principal that is a "member of the general public". It is used to aid in the attribution of rights to a "member of the general public" in some access control systems.

Credentials from both the ORBASEC SL2-GSSKRB and SL2-SSL have this attribute.:

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 UTF-8 encoding of a KerberosName that is the principal’s name, such as “bart@MYREALM.COM”.

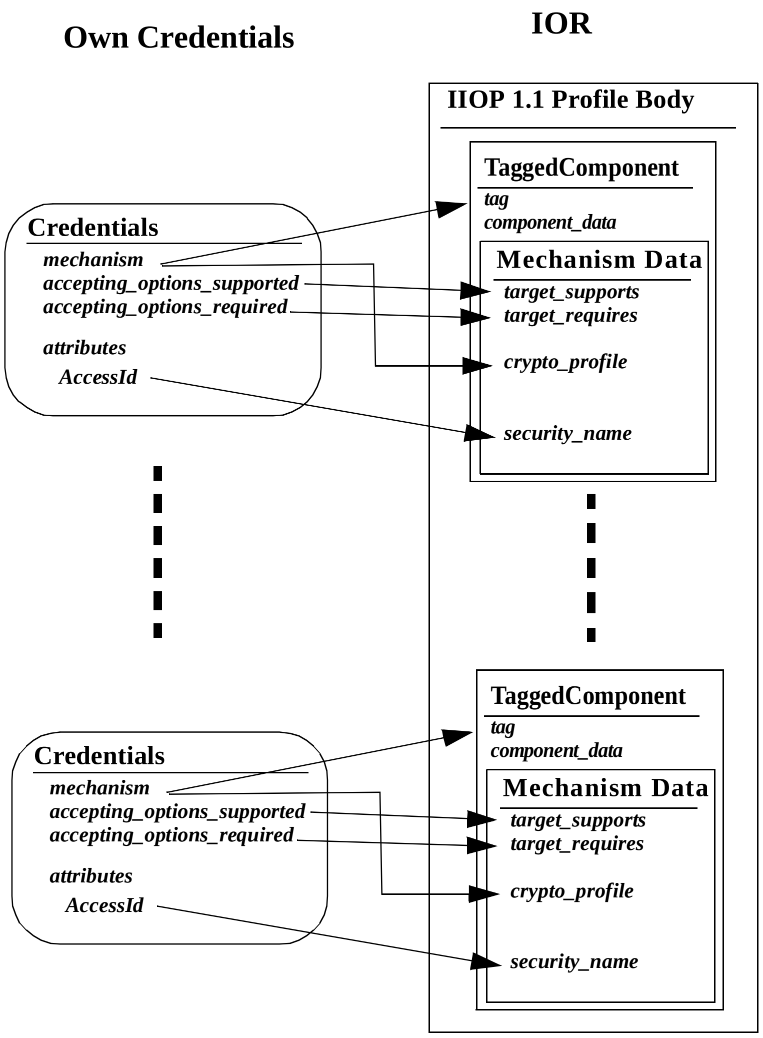
If you have the ORBASEC SL2-SSL distribution, the **value** field contains the UTF-8 encoding of a DirectoryName that is the principal’s name, which was found in the SubjectDN field of the prnicipal’s X.509 certificate. A string representation of such a structure might be “C=US, O=Adiron, OU=R&D, CN=Bart”. If using anonymous ciphers, the **value** field will contain the UTF-8 encoding of 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 “ORBAsec SL2 Extensions to SecurityManager” on page 76 and “ORBAsec SL2 Extensions to Current” on page 80] places a security component representing its capabilities and security name in the IIOP profile. The relationship between Credentials objects and the IIOP profile is depicted in Figure 2, “Mapping of Own Credentials Objects to IOR,” on page 148.

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. [6, 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 usually the value, depending on the mechanism, of the **AccessId** typed security attribute of the **Credentials.**

Note – Previous versions of ORBAsec SL2 used a different encoding scheme for the security name placed in the IOR. Therefore, IORs created with previous versions previous of ORBAsec SL2 are incompatible with ORBAsec SL2 2.1 and later.

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, and that at least one certificate in the chain is verified by one of the trusted authorities specified or taken as the default (See“The SSLIOP Authentication Model” on page 122). ORBASEC SL2-SSL also verifies that the certificate is 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();
```

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_policy_overrides** operation on the **PolicyCurrent** 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 94 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();
}
```

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():
}
```

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();
}
```

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();
}
```

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.

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 **PolicyCurrent** object's **set_policy_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 **PolicyCurrent** object may hold 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_policy_overrides** operation.

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.

If a particular policy is not set, then it is effectively a "don't care" for that policy type, and ORBAsec SL2 figures the most compatible Credentials and IOR components with respect to that policy type.

```
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[] { mechpol, delpol };

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

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 **PolicyCurrent** object's **get_policy_overrides** operation. An application programmer sets the default policies by setting them on the **PolicyCurrent** object by using its **set_policy_overrides** operation.

Since the policy override mechanism has not yet been standardized for **PolicyCurrent** 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_policy_overrides** and **set_policy_overrides** operations are found on the ORBASEC SL2 **SecLev2::PolicyCurrent** interface. See "Getting the PolicyCurrent Object" on page 83.

Setting policies on PolicyCurrent before a call to "impl_is_ready()" effectively sets the policies for the capsule. During the servicing of a request, setting policies on PolicyCurrent only lasts for duration of the thread. Be careful using this object and the threading model you choose.

```
// 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[] { mechpol, delpol };

orbasec.SecLev2.PolicyCurrent current =
                            // Get PolicyCurrent Object
current.set_policy_overrides(
                policies,
                  org.omg.CORBA.SetOverrideType.ADD_OVERRIDE);
```

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 63 for more details about how to initialize ORBASEC SL2 with your own SecurityReplaceable module.

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.

Note – The ORBAsec SL2 SSLIOP and IIOP modules do not have a similar notion of replaceability. Implementing SecurityReplaceable components only applies to SECIOP, whose specification is clearly outlined in the CORBA Security Specification [7].

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 implementor of a **SecurityLevel2::Credentials** object to ward off user mistakes and recognize bad arguments to parameters or attribute settings. Figure 3 on page 162, 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.

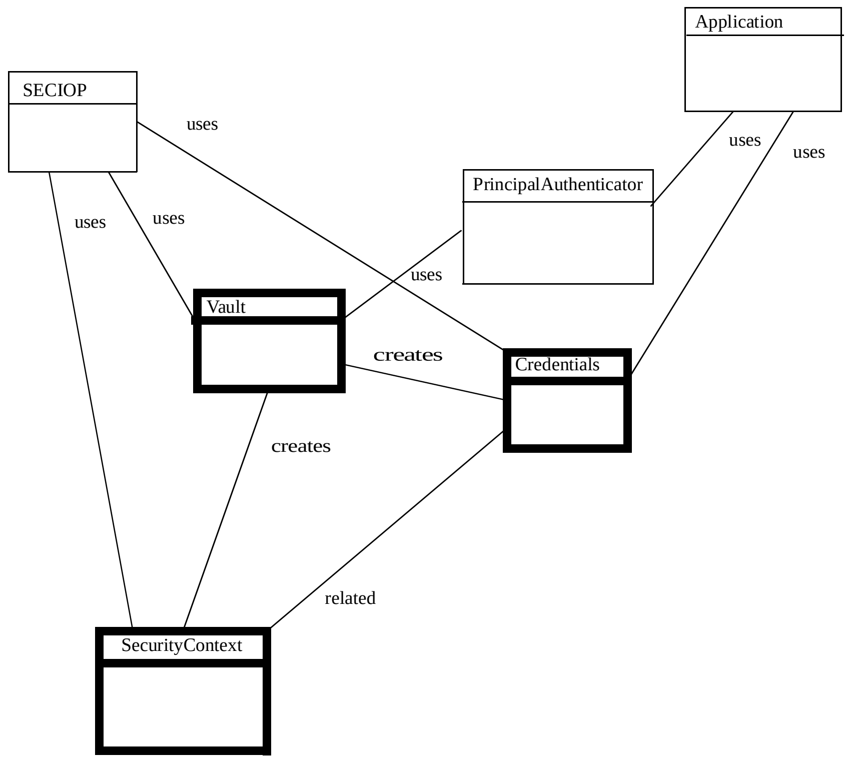


FIGURE 3. Security Replaceable Components

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::SecurityName          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,
  string                              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, [6, 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 **ClientSecurityContext** 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 [6, 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 do 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::SecurityName            security_name,
    in  any                               auth_data,
    in  Security::AttributeList           privileges,
    out SecurityLevel2::Credentials       creds,
    out any                               continuation_data,
    out any                               auth_specific_data
);

// Java
org.omg.Security.AuthenticationStatus
acquire_credentials(
    int                                 method,
    String                              mechanism,
    String                              security_name,
    org.omg.CORBA.Any                   auth_data,
    org.omg.Security.SecAttribute[]     privileges,
    org.omg.SecurityLevel2.CredentialsHolder
                                        creds,
    org.omg.CORBA.AnyHolder             continuation_data,
    org.omg.CORBA.AnyHolder             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 string 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 data held by the **CORBA::Any** 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 **SecurityManager** 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     any                          response_data,
    in     SecurityLevel2::Credentials  creds,
    out    any                          continuation_data,
    out    any                          auth_specfic_data
);
```

```
// Java
public org.omg.Security.AutenticationStatus
continue_credentials_aquision(
  org.omg.CORBA.Any                      response_data,
  org.omg.SecurityLevel2.Credentials     creds,
  org.omg.CORBA.AnyHolder                continuation_data,
  org.omg.CORBA.AnyHolder                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 the **SecurityManager** object'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 **SecurityManager** 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**.

```
// 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();
```

get_ior_components

This operation returns a list of IOP tagged components that go into the named profile for the Credentials.

```
// IDL
typedef sequaence<IOP::TaggedComponent> TaggedComponentList;
TaggedComponentList get_ior_components();

// Java
org.omg.IOP.TaggedComponent[]
get_ior_components();
```

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.

The **Credentials** interface is discussed in detail in “Credentials” on page 126. In this section, we discuss aspects of the **Credentials** interface that pertain to replaceability of security mechanisms within the SECIOP module, since the implementer will need to implement this interface.

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 may be one of the user accessible "own" **Credentials** on the **SecurityManager** 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::SecCredential-sType::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::SecAuth-Success** if the **Credentials** are fully initialized. It must have the value of **Secu-**

rity::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 mechanism parameter descriptions for the ORBAsec SL2 security mechanisms in “Principal Authenticator” on page 87 for details. 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);
```

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, you may have a need to define your own format for the value fields of SecAttibutes you provide. In such a case, you must supply an ASN.1 encoding of an OID representing a defining authority for the format you use. If the defining authority is left empty (byte [0]), SECIOP will assume the value fields are UTF-8 encodings of strings. See “Security Attributes of Credentials” on page 142 for more information.

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

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

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::OpaqueBuffer         refresh_data,
    out Security::OpaqueBuffer         out_token
  )


  boolean process_refresh_token(
    in Security::OpaqueBuffer         refresh_token
  )
  void discard_security_context(
    in  Security::OpaqueBuffer         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::OpaqueBuffer         refresh_data,
    out Security::OpaqueBuffer         out_token
);

// Java
public void refresh_security_context(
    org.omg.Security.OpaqueBuffer       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::OpaqueBuffer         refresh_data,
    out Security::OpaqueBuffer         out_token
);

// Java
public void discard_security_context(
   org.omg.Security.OpaqueBuffer       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::CredentialsList
                               client_credentials;
  readonly attribute Security::AssociationOptions
                               server_options_supported;
  readonly attribute Security::AssociationOptions
                               server_options_required;
  readonly attribute Security::SecurityName
                               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 cannot 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::SecurityName
                                server_security_name;
```

```
// Java
public String 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::SecurityName
                                 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::SecurityName
                                server_security_name;
```

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

CHAPTER 9

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 org.omg.CORBA.ORB init(
         String                argv[],
         java.util.Properties  properties
  );
```

```
public static org.omg.CORBA.ORB init(
       String                argv[],
       String                prefix,
       java.util.Properties  properties
);

public static org.omg.CORBA.ORB init(
       java.applet.Applet    applet,
       java.util.Properties  properties
);

public static org.omg.CORBA.ORB init(
       java.applet.Applet    applet,
       String                prefix,
       java.util.Properties  properties
);

public static org.omg.CORBA.BOA BOA_init(
       String                argv[],
       java.util.Properties  properties
);

public static org.omg.CORBA.BOA BOA_init(
       String                argv[],
       String                prefix,
       java.util.Properties  properties
);

public static org.omg.CORBA.BOA BOA_init(
       java.applet.Applet    applet,
       java.util.Properties  properties
);

public static org.omg.CORBA.BOA BOA_init(
       java.applet.Applet    applet,
       String                prefix,
       java.util.Properties  properties
);

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

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

public static org.omg.CORBA.ORB
orb();

public static org.omg.CORBA.BOA
boa();

public static java.util.Properties
get_properties();

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
  );
};
```

The ORB and BOA initialization methods of the SL2 class are discussed in detail in “ORBAsec SL2 Configuration” on page 54, as are the ORB, BOA, and Java Properties accessors.

Version

This field contains a string describing the version of ORBASEC SL2 that you are working with.

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 156].

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 156].

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.

CHAPTER 10 *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.

Note – The **orbasec.tools.CA** tool is made obsolete by the support for the Java KeyStore facility in ORBASEC SL2 2.1 and later. However, we have also supplied another tool, **KAC2KeyStore**, which will import the contents of a KeyAndCertificate file, supported in older versions of ORBASEC SL2, into a Java KeyStore.

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**
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 convenience 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 11

References

1. Blakely, Bob, “CORBA Security: An Introduction to Safe Computing with Objects”, The Addision-Wesley Object Technology Series, ISBN: 0201325659, October, 1999.
2. Kohl J, Neuman C., “The Kerberos Network Authentication Service (V5)”, Network Working Group RFC 1510, September 1993.
3. Java Cryptography Architecture and API Reference and Specification, http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html
4. The Object Management Group, “The Common Object Request Broker: Architecture and Sepcification”, Version 2.2, Feburary 1998.
5. The Object Management Group, “CORBAservices: Common Object Services Specification”, November 1997.
6. The Object Management Group, “Security Service Specification”, Version 1.2 Draft 4.1, 5 January, 1998.
7. The Object Management Group, “Security Service Specification”, Version 1.5, March 1999.
8. The Object Management Group, “Security Service Specification”, Version 1.7, January 2000.
9. Object Oriented Concepts, Inc. “ORBACUS C++ and Java”, Version 3.3, 2000.