Borland VisiBroker™ 8.0

Security Guide

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright 1992–2006 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

Microsoft, the .NET logo, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

VB 80 VisiBroker Security Guide
April 2007

Borland®

Contents

1

Getting Started with VisiSecure

As more businesses deploy distributed applications and conduct operations over the Internet, the need for a high quality application security has grown.

Sensitive information routinely pass over Internet connections between web browsers and commercial web servers; credit card numbers and bank balances are the two examples. For example, users engaging in commerce with a bank over the Internet must be confident that:

- They are in fact communicating with their bank's server, and not an imposter that mimics the bank for illegal purposes.
- The data exchanged with the bank will be unintelligible to network eavesdroppers.
- The data exchanged with the bank software will arrive unaltered. An instruction to pay $500 on a bill must not accidentally or maliciously become $5000.

VisiSecure lets the client authenticate the bank's server. The bank's server can also take advantage of the secure connection to authenticate the client. In a traditional application, once the connection has been established, the client sends the user's name and password to authenticate. This technique can still be used once a VisiSecure connection has been established, with the additional benefit that the user name and password exchanges will be encrypted. VisiSecure provides support for any number of authentication realms providing access to portions of distributed applications. In addition, with VisiSecure you can create authorization domains that delineate access-control rules for your applications.

VisiSecure overview

VisiSecure provides a framework for securing VisiBroker and AppServer. VisiSecure lets you establish secure connections between clients and servers.

VisiSecure design flexibility

Borland has designed VisiSecure to work with a variety of application architectures, so that it can support different types of current and future architectures. However, while VisiSecure represents a powerful security architecture, it cannot fully protect your servers by itself. You must be responsible for physical security, and configuring your base web server (host) and operating system services in the most secure manner possible.

Pluggability

VisiSecure allows many security technologies to be plugged in. Pluggability is provided at various levels. Security service providers can plug in and replace the entire set of security services and application developers can plug in smaller modules to achieve custom integration with their environment. The only layers which are not pluggable are the CSIv2 layer and the transport layer which are tightly integrated with the internal implementation of the VisiBroker ORB and interact heavily with each other.

VisiSecure for Java

VisiSecure is 100% Java and supports all security requirements of the J2EE 1.3 specification. VisiSecure uses the Java Authentication and Authorization System (JAAS) for authentication, the Java Secure Socket Extension (JSSE) for SSL communications, and the Java Cryptography Extension (JCE) for cryptographic operations. Most of the APIs for Java applications reflect the existing JDK or additional Java standard APIs. Care has been taken not to duplicate APIs at the different security layers. In some cases, VisiSecure feature set exceeds the J2EE 1.3 security requirements.

VisiSecure for Java features

VisiSecure has the following features:

- **Enterprise Java Beans (EJB) Container Integration:** VisiSecure seamlessly integrates EJB security mechanisms with the underlying CORBA Security Service and CSIv2. CORBA offers enhanced features to the security architecture of your bean. By utilizing VisiSecure, you have additional options over the relatively simple EJB security model.
- **Web Container Integration:** VisiSecure integrates with the web container by providing mechanisms to the web container that allow its own authentication and authorization engines to propagate security information to other EJB containers, as necessary. For example, a servlet trying to invoke an EJB container's bean will act on behalf of the original browser client that triggered the initial request. Security information supplied from the client will be propagated seamlessly to the EJB container. In addition, the web container authentication and authorization engine can be configured to use authentication LoginModules and authorization rolemaps supplied by Borland.
- **Security Services Administrator:** The administration and configuration of VisiSecure is performed using simple-to-use properties and supports tools like the Java keytool.
- **GateKeeper:** You can use GateKeeper to enable authenticated connections across a high-level firewall. This allows clients to connect to the server, even if the server and the application client are on opposite sides of a firewall. Use of the GateKeeper is fully documented in the *VisiBroker GateKeeper Guide*.
- **Secure Transport Layer:** VisiSecure utilizes SSL, the primary secure transport level communication protocol on the Internet, as a secure transport layer. SSL provides message confidentiality, message integrity, and certificate-based authentication support through a trust model.

VisiSecure for C++

VisiSecure for C++ offers similar feature as VisiSecure for Java. See “VisiSecure for C++ APIs” and “Security Properties for C++” for detailed information.

VisiSecure for C++ Features

VisiSecure for C++ has the following features:

- **Authentication and Authorization**: The Authentication and Authorization model are similar to VisiSecure for Java. This extends the capability of VisiSecure for C++ applications.
- **Security Services Administrator**: The administration and configuration of VisiSecure is performed using simple-to-use properties.
- **Secure Transport Layer**: VisiSecure utilizes SSL, the primary secure transport-level communication protocol on the Internet, as a secure transport layer. SSL provides message confidentiality, message integrity, and certificate-based authentication support through a trust model

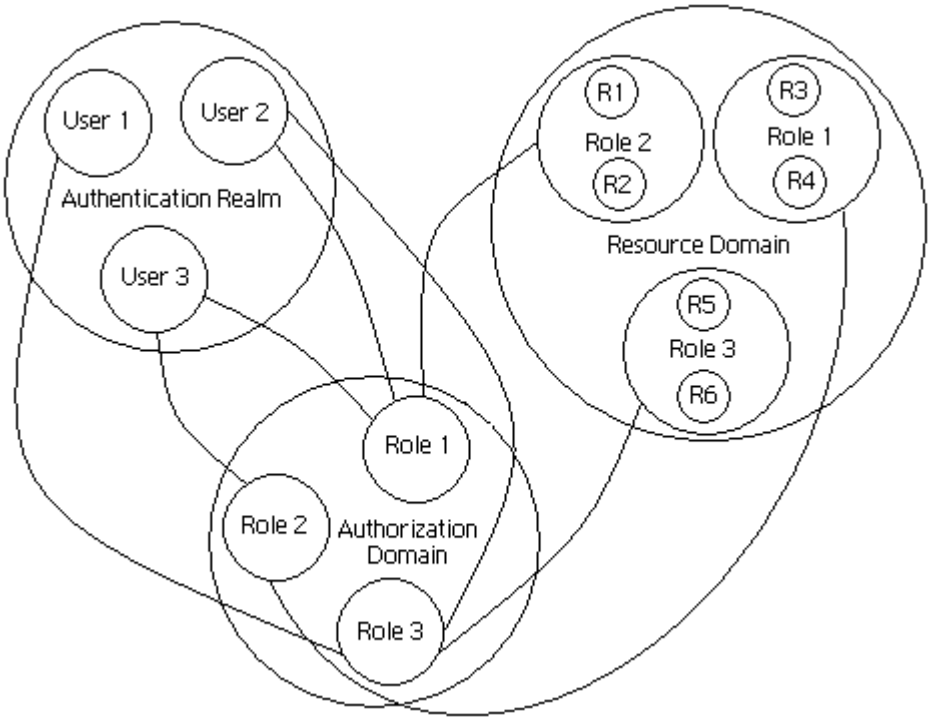
Basic security model

The basic security model describes VisiSecure and its components from a user's perspective. This is the logical model that VisiSecure users need to understand, configure and interact with. The security service groups entities of a system into the following three logical groups (domains):

- **Authentication realm (User domain):** This is simply a database of users. Each authentication realm describes a set of users and their associated credentials and Privileges attributes.
- **Resource Domain:** This represents a collection of resources of a single application. The application developer defines the access control policies for accessing resources in the application.
- **Authorization Domain:** This defines the set of rules that determine whether an access attempt to a particular resource is allowed.

The following figure displays the relationship among these domains.

Figure 1.1 Interaction Among Different Domains in VisiSecure



These three VisiSecure domains are closely related.

1. For authentication, you need an authentication realm. VisiBroker comes with a simple one, or you can use an existing supported realm, like an LDAP server.
2. For authorization, you need to set up roles, and associate users with those roles.
3. Then, you need to set up a resource domain, and grant access to the resources in that domain to certain roles.

Authentication realm (user domain)

An authentication realm is a database of users. Each authentication realm describes a set of users and their associated credentials and privileges, such as the user's password and the groups to which the user belongs, respectively.

Examples of authentication realms are: an NT domain, an NIS or yp database, or an LDAP server.

A “realm” represents a configuration entry that represents an authenticating target.

An authentication realm is defined both by the authentication mechanism such as LoginModules it uses, as well as a set of configuration options customized to point to the source of the data which contains the user information.

For example, if you are using LDAP, then the authentication realm specifies LDAP as the authentication protocol, specifies the name of the server, and specifies other configuration parameters. When you log on to a system, the system is authenticating you. For more information on setting up authentication realm, see “Setting up authentication realms.”

As *authentication realm* uses a single user LoginModules. It can be used with multiple user databases that support the same LoginModules. It allows the LoginModules to be independent of the actual user database

For example, if a vendor writes an authentication module to work with LDAP, that LoginModule can then be used to interact with different LDAP directories in different

environments, without having to rewrite or otherwise modify the authentication mechanism.

Setting up Resource domain

A *resource* defines an application component that VisiSecure needs to protect. VisiSecure organizes resources into *resource domains* containing every resource in an application. This means every remote method or servlet that is exposed by a server is essentially a resource.

The application developer defines access control policies for accessing resources in the application. These are defined in terms of roles. *Roles* provide a logical collection of permissions to access a set of resources. For more information, see “Authorization.”

In addition, applications may choose to be more security aware and provide access control for more fine grained resources such as fields, or access to external resources such as databases. The EJB and Servlet specifications provide standard deployment descriptor information that allow applications to define their access policies in terms of the set of roles required to access a given method.

Authorization domain

The authorization domain allows users to act in given roles. VisiSecure grants privileges to access resources based on these roles. When VisiBroker applications pass user identities from one application to another, the identity contains user information, and the permissions based on the specified roles. The caller's identity is then matched with the required rules to determine whether the caller satisfies the required rules. If the caller satisfies the rules, access is granted. Otherwise, the access is denied. For more information, on authorization domains, see “Authorization.”

Distributed Transmission

For a distributed environment, in addition to the three domains that make up the basic security model, the following must be considered:

- Distributed transmission of the authorization privileges
- Assertion and trusting assertion

The VisiSecure Service Provider Interface (SPI) provides interfaces and classes to address secure transportation, assertion, and assertion trust. The transmission (or interoperability) is handled by the underlying CSIv2 implementation. Since the implementation of the SPI is closely bundled with the VisiBroker ORB, it cannot be separated from the core as a generic SPI for other languages.

Specifically, the VisiSecure SPI classes enable customization of your Security Service in the following:

- Identification and Authentication
- Authorization (or access control decision making)
- Assertion trust

2

Authentication

The first layer of security protection for any system is authentication. This layer defines the process of verifying the entities who they claim to be. Most of the time, credentials are required to verify the identity of an entity.

VisiSecure employs the Java Authentication and Authorization Service (JAAS) framework to facilitate the interaction between the entities and the system. At the same time, the *authentication mechanism* concept is employed to represent the *format* (encoding and decoding process) for communicating or transporting authentication information between various components of the security subsystem.

Authentication is the process of verifying that an entity (human user, service, or component) is the one it claims to be. The authentication process includes:

1 acquiring credentials from the to-be-authenticated entity,

2 verifying the credentials.

Managing authentication with JAAS

The Java Authentication and Authorization Service (JAAS) defines extensions that allow pluggable *authorization* and user-based *authentication*. This framework effectively separates the implementation of authentication from authorization, allowing greater flexibility and broader vendor support. The fine-grained access control capabilities allow application developers to control access to critical resources at the granularity level that makes the most sense.

Basics of JAAS concepts

VisiSecure employs the Java Authentication and Authorization Service (JAAS) framework to facilitate the interaction between the entities and the system. Those who are new to the JAAS should familiarize themselves with the terms JAAS uses for its services. The concepts of *subjects*, *principals*, and *credentials* are of particular importance.

Subjects

JAAS uses the term *subject* to refer to any user of a computing service or resource. Another computing service or resource, therefore, is also considered a subject when it requests another service or resource. The requested service or resource relies on

names in order to authenticate a subject. However, different services may require different names in order to use them.

For example, your email account may use one username/password combination, but your ISP may requires a different combination. However, each service is authenticating the same subject—;namely yourself. In other words, a single subject may have multiple names associated with it. Unlike the example situation, in which the subject himself must know a set of usernames, passwords, or other authentication mechanisms at a specific time, JAAS is able to associate different names with a single subject and retain that information. Each of these names is known as a *principal*.

Principals

A *principal* represents any name associated with a subject. A subject could have multiple names, potentially one for each different service it needs to access. A subject, therefore, comprises a set of principals, such as in the code sample below:

Java

```
public interface Principal {
   public String getName();
}
public final class Subject {
   public Set getPrincipals()
}
```

C++

```
class Principal {
    public:
       std::string getName() const=0;}
class Subject {
    public:
    Principal::set& getPrincipals();
}
```

Principals populate the subject when the subject successfully authenticates to a service. You do not have to rely on public keys and/or certificates if your operational environment has no need for such robust technologies.

To return the principle name(s) for a subject from the application context, use `getCallerPrincipal`.

Note

Principals participating in transactions may not change their principal association within those transactions.

Credentials

In the event that you want to associate other security-related attributes with a subject, you may use what JAAS calls *credentials*. Credentials are generic security-related attributes like passwords, public-key certificates, and such. Credentials can be any type of object, allowing you to migrate any existing credential information or implementation into JAAS. Or, if you want to keep some authentication data on a separate server or other piece of hardware, you can simply store a reference to the data as a credential. For example, you can use JAAS to support a security-card reader.

Public and private credentials

Credentials in JAAS come in two types, public and private. *Public* credentials do not require permissions to access them. *Private* credentials require security checks. Public credentials could contain public keys, and such, while private credentials are private keys, encryption keys, sensitive passwords, and such. Consider the following subject:

Java

```
public final class Subject {
   ...
   public Set getPublicCredentials()
}
```

C++

```
class Subject {
    public:
    Credential::set& getPrivateCredentials();
}
```

No additional permissions would be necessary to retrieve the public credentials from the subject, except in the case:

Java

```
public final class Subject {
   ...
   public Set getPrivateCredentials()
}
```

C++

```
class Subject {
    public:
    Credential::set& getPrivateCredentials();
}
```

For Java, permissions are required for code to access the private credentials in a Subject. For c++, all codes are local and are therefore trusted.

Credential identifies an identity for authentication or for verifying. It also determines the identity and its associated roles. No permission is required to access both public and private credentials.

Public credentials is used for authorization only. Private credentials are used for caching purposes. These credentials are populated by login modules.

For more information on permissions in Java, see the JAAS Specification from Sun Microsystems.

Authentication and pluggability

Authentication in VisiBroker is a JAAS implementation allowing pluggable authentication. The JAAS logon service separates the configuration from implementation. A low-level system programming interface called the LoginModule, provides an anchor point for pluggable security modules.

At the same time as system identification, the *authentication mechanism* concept is employed to represent the “format” for communicating (or transporting) authentication information between various components of the security subsystem. The security service provider for the authentication/identification process implements the specific format (encoding and decoding process) that is to be used by the underlying core system.

In a distributed environment, the authentication process is further complicated by the fact that the representation of the entity and the corresponding credential must be transported among peers in a generic fashion. Therefore, the VisiSecure Java SPI employs the concept of the `AuthenticationMechanism` and defines a set of classes for doing authentication/identification in a distributed environment.

Identity, trust and authentication

Relationship between trust and authentication

Authentication is a process of verifying an identity. When the verification is succesful, the identity becomes a trusted identity. In other words: a successful authentication of an identity puts trust on the identity. Trust is a result of a successful authentication. It is also the result of the identity assertion.

If identity A is successfuly authenticated and therefore trusted. But later, identity A asserts identity B and B is therefore trusted although we never verify that B of its identity. This is because the system trusts A and all that A asserts.

Trust can be applied at the transport level if a certificate identity is presented, or at even higher levels (at the CSIv2 layer) where the identity takes the form of a username/password.

Java

For trusting certificates with Java code, VisiSecure provides mechanisms to support user-provided JSSE `X509TrustManager` that indicates whether a given certificate chain is trusted. You can also specify a Java keystore where certificate entries are trusted using standard Java properties.

C++

For VisiBroker for C++ users, the a set of APIs that allow trustpoints (trusted certificates) to be configured is provided as well. For more information, see "VisiSecure for C++ APIs."

Note: For certificate authentication, login modules cannot be used.

Identities

Any system that needs to engage in secure communication as a client must be configured to have an identity that represents the user/client on whose behalf it is acting. When using SSL with mutual authentication, a server also needs a certificate to identify itself to the client.

In addition to many clients and users that need to be authenticated to the various VisiBroker services, the VisiSecure itself needs to be provided with its own identity. This allows the server to identify itself when it communicates with other secure servers or services. It also allows end-tier servers to trust assertions made by this server in the case where this server acts on behalf of other clients.

System identity

Any system first needs to identify itself before being allowed access to resources. Client identification is always required for resource access. In a CORBA/J2EE environment, the need for identification also exists for servers as well. Servers need identification in two cases:

- One, when using SSL for transport layer security, the server typically needs to identify itself to the client.
- Two, when mid-tier servers make further invocations to other mid-tier or end-tier servers, they need to identify themselves before being allowed (potentially) to act on behalf of the original caller.

Client identity

There are situations where the client process does not have any information on the realm that it needs to authenticate against. In this case, by default, the client consults the server's IOR for a list of available realms, and the user is given the option to choose one to which to supply username and password. This username/password will be used by the server, which will consult its configuration file for the specified realm, and use the information collected from the client as data for its specified `LoginModule`.

For example, if the following is the server side configuration file, then the information collected or entered by a user will be used for its `JDBCLoginModule`.

```
SecureRealm{
   com.borland.security.provider.authn.JDBCLoginModule required
   DRIVER=F"com.borland.datastore.jdbc.DataStoreDriver"
   URL="jdbc:borland:dslocal:../userdb.jds"
   USERNAMEFIELD="USERNAME"
   GROUPNAMEFIELD="GROUPNAME"
   GROUPTABLE="UserGroupTable"
};
```

The default behavior of the process can be changed through properties. You can set the retry count by setting `vbroker.security.authentication.retryCount`. The default is `3`. You can change and set the new retry count.

The security properties including those for authentication are listed and described in the "Security Properties for Java" and "Security Properties for C++."

Configuring authentication

The `authentication config` file contains the data necessary to authenticate a user to one or more realms and defines an authentication mechanism and provides the code to interact with a specific type of authentication mechanism (for example, LoginModules).

The configuration must specify which LoginModules should be used for a particular application, and in what order the LoginModules should be invoked. For more information, refer to "Associating a LoginModule with a realm".

An example of `config.jaas` file looks like this:

```
customrealm {
 CustomLoginModule required;
};
```

This defines a realm called `customrealm`, which requires the use of the `custom loginModule`.

A login configuration contains the following information. Each realm entry has a particular syntax that must be followed. For more information on realm syntax, refer to "Syntax of a realm entry".

Each entry in the configuration is indexed via realm name and contains a list of LoginModules configured for that application.

Each LoginModule is specified via its fully qualified class name. Authentication proceeds down the module list in the exact order specified. If an application does not have specific entry, it defaults to the specific entry.

The Flag value controls the overall behavior as authentication proceeds down the stack. For description of the valid values for Flag and their respective semantics, refer to "Syntax of a realm entry".

For information on LoginModules, see "Creating LoginModules."

Authentication property settings

Whether it is server or client authentication and whether it is done using public-key certificates or passwords, it is determined by property settings. For more information, see "Security Properties for C++" and "Security Properties for Java."

The security configuration uses properties and a configuration file to define the identities that represent the system. This configuration file is populated with all the LoginModules necessary for authentication to the various realms to which this process needs to authenticate.

For example:

Set the property `vbroker.security.login=true`
Set the property `vbroker.security.login.realms=payroll,hr`
Set the following realm information in a file reference by `vbroker.security.authentication.config=<config-file>`
Set the property `vbroker.security.callbackhandler=<callback-handler>`

In the `<config-file>` setup the following:

```
payroll {
com.borland.security.provider.authn.HostLoginModule required;
};

hr {
  com.borland.security.provider.authn.BasicLoginModule required
 DRIVER=com.borland.datastore.jdbc.DataStoreDriver
 URL="jdbc:borland:dslocal:../userdb.jds"
 TYPE=BASIC
 LOGINUSERID=admin
 LOGINPASSWORD=admin;
};
```

In this code sample:

- The process will already know something about the realms to which it needs to authenticate through the property `vbroker.security.login.realms`.
- The process knows it will authenticate to the host on which it is running (logically representing the “payroll” realm), and so sets itself up to invoke this LoginModule.
- The process also knows that it must log into the “hr” realm, and so sets up a LoginModule to this end as well.

The format of the realm information passed into `vbroker.security.login.realms` is as follows:

```
<authentication Mechanism>#<Authentication Target>
```

This format is called *Formatted Target*.

Formatted target

Formatted Target is the means of representing authentication mechanism.

A formatted target is of the form:

```
<authentication mechanism>#<mechanism specific target name>
```

For example:

```
Realm1, Realm3, GSSUP#Realm4,
```

In the above example, realm1, realm3 and realm4 are specific instances of GSSUP authentication mechanism. (By default, if you don’t specify the authentication mechanism, it is assumed to be GSSUP.)

This can be used to represent how LoginModules communicate with the authentication mechanism and how the mechanism on one process communicates with an equivalent mechanism on another process. The mechanism specific target name represents how the mechanism represents this target.

For more information on authentication mechanism, refer to “Authentication mechanisms”.

Setting the config file for client authentication

Each process uses its own configuration file containing the configuration for the set of authentication realms that the system supports for client authentication.

To set the location of the configuration file:

1. Set the `vbroker.security.authentication.config` property to the path of the configuration file.

2. If desired, you can specify more than one login configuration file as follows:

```
vbroker.security.authentication.configs=myconfig, yourconfig, hisconfig, herconfig
vbroker.security.authConfig.myconfig=<the physical file name for myconfig>
vbroker.security.authConfig.yourconfig=<the physical file name for yourconfig>
vbroker.security.authConfig.hisconfig=<the physical file name for hisconfig>
vbroker.security.authConfig.herconfig=<the physical file name for herconfig>
```

If more than one login configuration file is specified then the files are read and concatenated into a single configuration.

Note: the use of forward or backward slashes are based on the underlying file system. The URLs always use forward slashes, regardless of what operating system the user is running.

Setting up authentication realms

A system administrator determines the authentication technologies, or Login Modules, to be used for each application and configures them in the configuration file.

Following are the steps to setup the authentication realm:

1. Create an authentication configuration file containing one or more realms. For information on Creating configuration file, refer to “Configuring authentication”.
2. Use the property `vbroker.security.authentication.config` to involve the configuration file into the runtime.

```
vbroker.security.authentication.config=<the filename of the config file>
```

An example:

```
customrealm {
 CustomLoginModule required;
};
```

In the above example, the realm entry is named "customrealm". This name must be unique as it will be used by the VisiSecure to refer this entry. The entry specifies the LoginModule to be used for the user authentication.

This LoginModule is “required” for the authentication to be considered successful. The LoginModule will succeed only if the name and password supplied by the user are successfully used to log the user into the system.

For setting up the configuration file for client and server, refer to the basic authentication example in the \\Borland\VisiBroker\examples\vbroker\security\basic folder. The example given here has all the basic setting needed to secure the application.

Different types of Authentication

With the VisiBroker implementation of JAAS, you can set up different mechanisms of authentication. You can have server authentication, where servers are authenticated by clients using public-key certificates. You can also have client authentication. Clients can be authenticated using passwords or public-key certificates. That is, the server can be configured to authenticate clients with a password or clients with public-key certificates. Whether it is server or client authentication and whether it is done using public-key certificates or passwords, it is determined by property settings. For more information see “Authentication property settings”

Servers

Authentication can be accomplished using a standard username/password combination. For servers using username and password, authentication is performed locally since the realms are always known.

There can be constraints on certificate identities, depending on whether they are stored in a KeyStore or whether they are specified through APIs.

Clients

To authenticate clients using usernames and passwords, several things need to happen. The server should expose a set of realms to which it can authenticate a client. Each realm should correspond to a JAAS LoginModule that actually does the authentication. Finally, the client should provide a username and password, and a realm under which it wishes to authenticate itself.

For clients using usernames and passwords, there can be constraints about what the client knows about the server's realms. Clients may have prior knowledge of the server's supported realms or none at all at the time of identity inquiry.

Client always authenticates at the server end for which the client has to do the identity enquiry.

If the client does not know about the server’s realm upfront, then it has to read the server IOR and reactively do an identity enquiry to make the server authenticate.

Client can authenticate itself if the server’s realm is known upfront. Even in such case, server will authenticate again.

Keeping these constraints in mind, the VisiSecure supports the following usage of authentication models: “GSSUP mechanism” and “Certificate mechanism”. You can use any of these to provide an identity to the server or client.

Authentication mechanisms

An *authentication mechanism* represents the encoding/decoding format for communicating authentication information between various components of the security subsystem.

For example, it represents how LoginModules communicate with the mechanism and how the mechanism on one process communicates with an equivalent mechanism on another process.The mechanism specific target name represents how the mechanism represents this target.

There are two types of authentication mechanisms supported by VisiSecure:

- GSSUP Mechanism
- Certificate Mechanism

GSSUP mechanism

VisiSecure provides a mechanism for a simple username/password authentication scheme. This mechanism is called *GSSUP*. The OMG CSIv2 standard defines the interoperable format for this mechanism. The LoginModule to mechanism interaction model is defined by Borland. This is because the mechanism implementation needs to translate the information provided by a LoginModule to information (to a specific format) it can transmit over the wire using CSIv2.

As mentioned above, the target name for a mechanism is specific to that mechanism. For the GSSUP mechanism, the target name is a simple string representing a target realm (for example, in the JAAS configuration file, on the receiving tier). So, if a server has a configuration file with one realm defined, for example “ServerRealm”, a client side representation of this realm would be:

```
GSSUP#ServerRealm
```

Note

For convenience, since the GSSUP mechanism is always available in VisiBroker, you can omit the “GSSUP#” from the target name. However, this is only for the GSSUP mechanism. When the security service interprets a “realm” name, it first attempts to resolve the realm name with a local JAAS configuration entry. If that fails, it treats that realm name as representing “GSSUP#”.

GSSUP based authentication:

- “Authenticating clients with usernames and passwords”
- “Username/password authentication using APIs”

Authenticating clients with usernames and passwords

For authentication, you need username/password or certificates. Username/password and certificates can be collected from user through JAAS callback handlers. These can also be collected through APIs.

Username/password authentication using LoginModules for known realms

If the realm to which the client or server wishes to authenticate is known, the client-side can be configured by setting properties as below:

```
vbroker.security.login=true
vbroker.security.login.realms=<known-realm>
```

Username/password authentication using APIs

For C++:

The following code sample demonstrates the use of the login APIs. This case uses a wallet. For a full description of the four login modes supported, see "VisiSecure for C++ APIs" and "Security SPI for C++."

```
int main(int argc, char* const* argv) {
    // initialize the ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->resolve_initial_references("VBSecurityContext");
    Context* c = dynamic_cast<Context*> (obj.in());
// Obtain a walletFactory
    CORBA::Object_var o = orb->resolve_initial_references("VBWalletFactory");
    vbsec::WalletFactory* wf = dynamic_cast<vbsec::WalletFactory*>(o.in());
    vbsec::Wallet* wallet = wf->createIdentityWallet( <username>, <password>, <realm>);
    c->login(*wallet);
}
```

For Java:

The following code sample demonstrates the use of the login APIs. This case uses a wallet. For a full description of the four login modes supported, go to the VisiSecure for Java API and SPI sections.

```
public static void main(String[] args) {
   //initialize the ORB
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
   com.borland.security.Context ctx = (com.borland.security.Context)
      orb.resolve_initial_references("VBSecurityContext");
   if(ctx != null) {
      com.borland.securty.IdentityWallet wallet =
         new com.borland.security.IdentityWallet(<username>,
            <password>.toCharArray(), <realm>);
      ctx.login(wallet);
   }
}
```

Certificate mechanism

The *Certificate mechanism* is a mechanism that is used for identification using certificates. This mechanism is different from GSSUP; Certificates are used instead of username/password, and these identities are used at the SSL layer and not at the higher CSIv2 over IIOP layer.

You can put certificates into VisiSecure using certificate login or wallet APIs. When using wallet APIs, you need to specify the usage through the constant definitions in the `vbsec.h` file, class `vbsec::WalletFactory`. For more information, see "class vbsec::WalletFactory".

For certificate based authentication using Java, see "Certificate based authentication using Certificate wallet".

Using certificate login, you need to specify the target realm using the following format:

```
Certificate#<target>
```

Note

If you do not specify the usage, the default is `ALL`.

The following describes the available targets defined for the certificate login mechanism.

Target	Description
Certificate#CLIENT	Identifies this process in a client role. When a user establishes an identity for this target, the certificate identity established will be used when this process acts as a client. In other words, this certificate will identify this process when it establishes outgoing SSL connections.
Certificate#SERVER	Identifies this process in a server role. When a user establishes an identity for this target, this process will use the certificate identity established to identify itself when it is accepting SSL connections.
Certificate#ALL	Identifies this process in all roles. This identity is used in both of the above roles.

A process can have either a client and server identity that are different or an identity that is used in all roles, but not both. In other words, you cannot establish an identity in the `Certificate#CLIENT` and the `Certificate#ALL` targets simultaneously.

Note

For backward compatibility, wallet properties and SSL APIs are supported; certificate identities established this way are only treated as `Certificate#ALL`.

VisiSecure includes several common LoginModules for server and client authentication as well as the Security Provider Interface classes for Java and C++ that enable you to “plug-in” security service provider implementations of authentication and identification.

Certificate based authentication:

- “Certificate-based authentication using KeyStores through property settings:”
- “Certificate-based authentication using APIs”
- “Certificate based authentication using APIs with pkcs12Server”
- “Certificate based authentication using Certificate wallet”
- “pkcs12-based authentication using APIs”

Certificate-based authentication using KeyStores through property settings:

This section demonstrates how to make a minimal SSL configuration on the simplest, non-security aware VisiBroker example such that client and server communicate using SSL that in turn enable them to perform mutual PKI authentication. Only executables from basic/bank_agent are re-used to emphasize that in order to secure non-security aware application no source code changes are required.

1. Firstly, please familiarize with the C++ version of the simplest VisiBroker examples residing in the directory examples/basic/bank
2. Copy over only the executables of the Server (Server.exe on windows) and Client (Client.exe on windows) to this directory.
3. Make sure that osagent is up and running as usual
4. Launch server using the command below:
   ```
   prompt> Server -DORBpropStorage=cpp_server.properties -
   Dvbroker.orb.dynamicLibs=<VisiSecure shared library name>
   ```
5. Launch client using the command below:
   ```
   prompt> Client -DORBpropStorage=cpp_client.properties -
   Dvbroker.orb.dynamicLibs=<VisiSecure shared library name>
   ```
6. Open property files **cpp_server.properties, cpp_client.properties** and notice how the certificates and private keys are installed using wallet property set in that file.

C++ Server properties

```
vbroker.security.peerAuthenticationMode=require_and_trust
vbroker.security.requireauthentication=false

vbroker.security.trustpointsRepository=Directory:./trustpoints
vbroker.security.server.transport=SECURE_ONLY

vbroker.security.wallet.type=Directory:./identities
vbroker.security.wallet.identity=frans
vbroker.security.wallet.password=frans
```

For description about these properties, refer to "Security Properties for Java".

C++ client properties

```
vbroker.security.trustpointsRepository=Directory:./trustpoints
vbroker.security.peerAuthenticationMode=require_and_trust

vbroker.security.wallet.type=Directory:./identities
vbroker.security.wallet.identity=charles
vbroker.security.wallet.password=charles
```

For description about these properties, refer to "Security Properties for C++".

7 Browse through the content of subdirectory **identities** and **trustpoints** and understand how the directory wallet and trustpoints are structured.

Note:

VisiSecure shared library name depends on the platforms.

For example:

on win32, it is vbsec.dll,
on Solaris 64 bit, it is vbsec64.so,
on HPUX 64 bit std build, it is vbsec64_p.sl.
It is recommended that you check your ${VBROKER_DIR}/lib directory

Certificate-based authentication using APIs

1 Build the example as mentioned in the basic/bank_SSL example in the example folder by executing the command:
`nmake cpp` (for windows) or
`make cpp` (for unix).

When build successfully, there are executables created
`SecureServer.exe` on windows and `SecureClient.exe` on windows.

2 Make sure osagent is up and running.

3 Launch server using the command below:
`prompt> SecureServer`

4 Launch client using the command below:
`prompt> SecureClient`

5 Launch either server or client or both using `-Dvbroker.app.useCRL=true`, and notice how the mutual SSL authentication fails and client gets `NO_PERMISSION` exception.
For example,
`prompt> SecureClient -Dvbroker.app.useCRL=true`

6 Read and learn from `SecureServer.C, SecureClient.C`
- how they perform the security initialization in their `main()` and after `ORB_Init()`.
- how they impose `peerAuthenticationMode=require_and_trust` and `alwaysSecure=true` through QoP

SecureServer.C

...

```
if (ssp) {
      CORBAsec::ASN1ObjectList chain;
      chain.length( bank::numberOfCertificates);
      CORBA::ULong L;
      for (CORBA::ULong i = (CORBA::ULong)0; i < bank::numberOfCertificates; i++) {
        L = (CORBA::ULong) strlen( bank::certificate[i]);
        chain[i].replace ( L, L, (CORBA::Octet*)bank::certificate[i],
                          (CORBA::Boolean)false );
      }            // Wrap the b64 certificate chain in an ASN1ObjectList as
                   // required by the certificate factory

      CORBAsec::X509CertList_var certchain =
               ssp->getCertificateFactory().importCertificateChain( chain);
                   // Consult the certificate factory to convert the chain
                   // into an X509CertList as required to create an SSLContext
```

Note:

In the resulting list, the order is reversed: The root cert is list[0]

```
      L = (CORBA::ULong) strlen( bank::privateKey);
      CORBAsec::ASN1Object key ( L, L, (CORBA::Octet*)bank::privateKey,
                                 (CORBA::Boolean)false );
                   // Wrap the b64 private key in an ASN1Object

      CORBAsec::ASN1Object_var privatekey =
               ssp->getCertificateFactory().importPrivateKey( key);
                   // Consult the certificate factory to convert the private key
                   // into a DER wrapped inside an ASN1Object

      const char* const sword = "frans";
      L = (CORBA::ULong) strlen( sword);
      vbsec::VBSSLContext* sslctx = ssp->createSSLContext (
        *certchain,
        *privatekey,
        CSI::UTF8String( L, L, (CORBA::Octet*)sword, (CORBA::Boolean)false )
      );           // Consult the SecureSocketProvider to create an SSLContext
                   // from the chain and the private key.

      CORBAsec::X509Cert* cacert = (*certchain)[(CORBA::ULong)0];
      sslctx->addTrustedCertificate( *cacert);
```

The root of this chain is implicit part of trustpoint. But it does not happen automatically

For Java:

If you do not want to use KeyStores directly, you can specify certificates and private keys using the `CertificateWallet`API. This class also supports the pkcs12 file format.

```
X509Certificate[] certChain = ...list-of-X509-certificates...
PrivateKey privKey = private-key
com.borland.security.CertificateWallet wallet =
   new com.borland.security.CertificateWallet(alias,
      certChain, privKey, "password".toCharArray());
```

The first argument in the new Certificate wallet is an alias to the entry in the KeyStore, if any. If you are not using keystores, set this argument to `null`.

Certificate based authentication using APIs with pkcs12Server

This section demonstrates how to use VisiSecure API for handling of a PKCS12 storage, a very widely acceptable storage format for certificates and private keys.

1 Build the example as mentioned in the bank_SSL example in the example folder by executing the command:
`nmake cpp` (for windows) or
`make cpp` (for unix).

When build successful, there will be executables created `pkcs12Server.exe` on windows

2 Make sure osagent is up and running

3 Launch server using the command below:
```
prompt> pkcs12Server frans.pfx frans
```

4 Launch client using the command below:
```
prompt> SecureClient
```

5 Launch client using `-Dvbroker.app.useCRL=true`, and notice how the mutual SSL authentication fails and client gets `NO_PERMISSION` exception.
```
prompt> SecureClient -Dvbroker.app.useCRL=true
```

6 Read and learn from `pkcs12Server.C`
Notice how it installs certificates and a private key from a PKCS12 file.

PKCS12Server.C

```
...
   if (ssp) {
         CORBA::ULong L = (CORBA::ULong) bank::BUF_SIZE;
         CORBAsec::ASN1Object pkcs12bytes( L, L, bank::gBuffer, (CORBA::Boolean)0 );

         L = (CORBA::ULong) strlen( argv[2]);
         CSI::UTF8String sword( L, L, (CORBA::Octet*)argv[2], (CORBA::Boolean)0 );
         CORBAsec::X509CertList_var certchain =
           ssp->getCertificateFactory().importCertificateChain ( pkcs12bytes,
                                                                  sword );
                     // Consult the certificate factory to convert the chain
                     // into an X509CertList as required to create an SSLContext
```

It is IMPORTANT to note that in the resulting list, the order is reversed. The root cert is list[0]

```
         CORBAsec::ASN1Object_var privatekey =
                  ssp->getCertificateFactory().importPrivateKey( pkcs12bytes,
                                                                 sword );
                     // Consult the certificate factory to convert the private key
                     // into a DER wrapped inside an ASN1Object

         if ( !certchain || !privatekey ) {
           cerr << "Fail to import certificates and private key from pkcs12 "
                << "file named: " << argv[1] << endl;
           exit( 1);
         }
         vbsec::VBSSLContext* sslctx = ssp->createSSLContext (
           *certchain,
           *privatekey,
           sword
         );          // Consult the SecureSocketProvider to create an SSLContext
                     // from the chain and the private key.
```

Java only

Certificate based authentication using Certificate wallet

Create a new wallet

```
         com.borland.security.provider.CertificateWallet wallet =
             new com.borland.security.provider.CertificateWallet (null, certChain,
                encryptedPrivateKey.getBytes (), "Delt@$$$".toCharArray());
```

Get the security context:

```
      // Login
         com.borland.security.Context ctx = (com.borland.security.Context)
             orb.resolve_initial_references ("VBSecurityContext");
         ctx.login (wallet);
```

Pass the wallet that was initially created in step 1

```
   c->login(*wallet);
   }
```

Setting wallet properties:

You can set the transport level security by setting the properties below:

Use these properties below to set the transport identity for SSL

```
vbroker.security.wallet.type=Directory:./identities
vbroker.security.wallet.identity=frans
vbroker.security.wallet.password=frans
```

pkcs12-based authentication using KeyStores

You can use the same APIs discussed in "Username/password authentication using APIs" to login using pkcs12 KeyStores. The realm name in the IdentityWallet should be `CERTIFICATE#ALL`, the `username` corresponds to an alias name in the default KeyStore that refers to a Key entry, and the `password` refers to the password needed to unlock the pkcs12 file. The property `javax.net.ssl.KeyStore` specifies the location of the pkcs12 file.

pkcs12-based authentication using APIs

If you do not want to use KeyStores directly, you can import certificates and private keys using the `CertificateFactory`API. This class also supports the pkcs12 file format.

```
CORBA::Object_var o = orb->resolve_initial_references("VBSecureSocketProvider");
vbsec::SecureSocketProvider* ssp = dynamic_cast<vbsec::SecureSocketProvider*>(o.in());

const vbsec::CertificateFactory& cf = ssp->getCertificateFactory ();
```

The first argument in the new Certificate wallet is an alias to the entry in the KeyStore, if any. If you are not using keystores, set this argument to `null`.

Creating LoginModules

A *LoginModule* defines an authentication mechanism and provides the code to interact with a specific *type* of authentication mechanism. Each LoginModule is customized using authentication options that points it to a specific data source and provide other customizable behavior as defined by the author of the LoginModule.

Each LoginModule authenticates to a particular authentication realm (any authenticating body or authentication provider- for example, an NT domain). An authentication realm is represented by a configuration entry in a JAAS configuration file. A JAAS configuration entry contains one or more LoginModule entries with associated options to configure the realm. For more information, see "Associating a LoginModule with a realm".

LoginContext class and LoginModule interface

VisiSecure uses the class `LoginContext` as the user API for the authentication framework. The `LoginContext` class uses the JAAS configuration file to determine which authentication service has to be plugged-in under the current application.

For Java

```
public final class LoginContext {
public LoginContext(String name)
public void login()
   public void logout()
   public Subject getSubject()
}
```

For C++

```
class LoginContext{
    public:
        LoginContext(const std::string& name, Subject *subject=0, CallbackHandler
*handler=0);
        void login();
        void logout();
        Subject &getSubject() const;
}
```

The authentication service itself uses the LoginModule interface to perform the relevant authentication.

For Java

```
public interface LoginModule {
   boolean login();
   boolean commit();
   boolean abort();
   boolean logout();
}
```

for C++

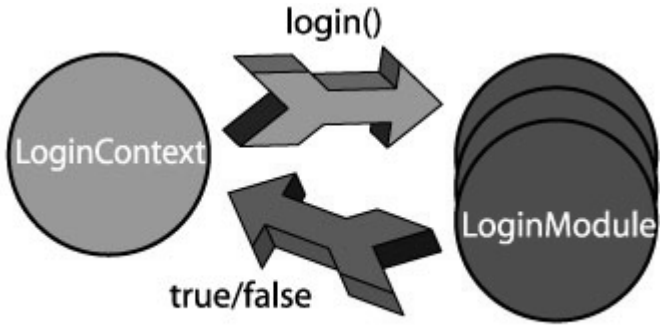
```
class LoginModule {
    public:
        virtual bool login()=0;
        virtual bool logout()=0;
        virtual bool commit()=0;
        virtual bool abort()=0;
}
```

It is possible to stack LoginModules and authenticate a subject to several services at one time.

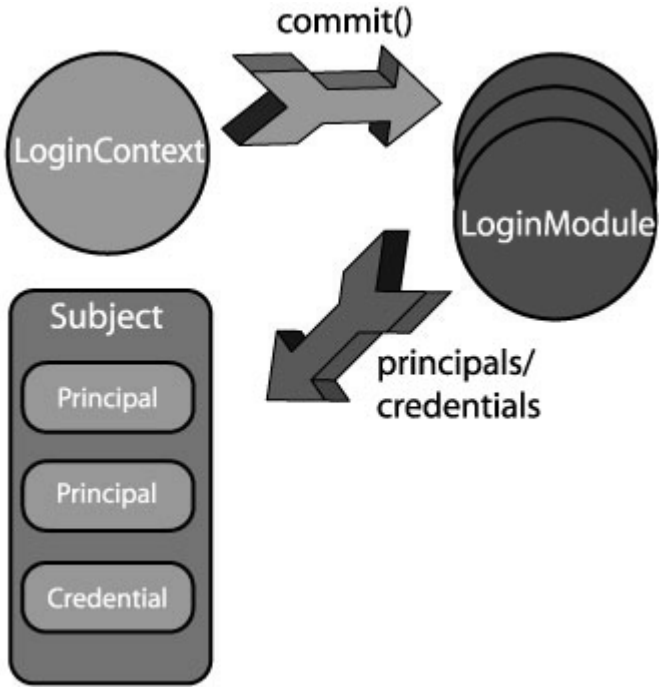
Authentication and stacked LoginModules

Authentication proceeds in two phases in order to assure that all stacked `LoginModules` succeed (or fail, otherwise).

1 The first phase is the “login phase,” during which the `LoginContext` invokes `login()` on all configured `LoginModules` and instructs each of them to attempt authentication.



2 If all necessary LoginModules successfully pass, the second, “commit phase” begins, and `LoginContext` calls `commit()` on each LoginModule to formally end the authentication process. During this phase the LoginModules also populate the subject with whatever credentials and/or authenticated principals are necessary for continued work.



Note

If either phase fails, the `LoginContext` calls `abort()` on each `LoginModule` and ends all authentication attempts.

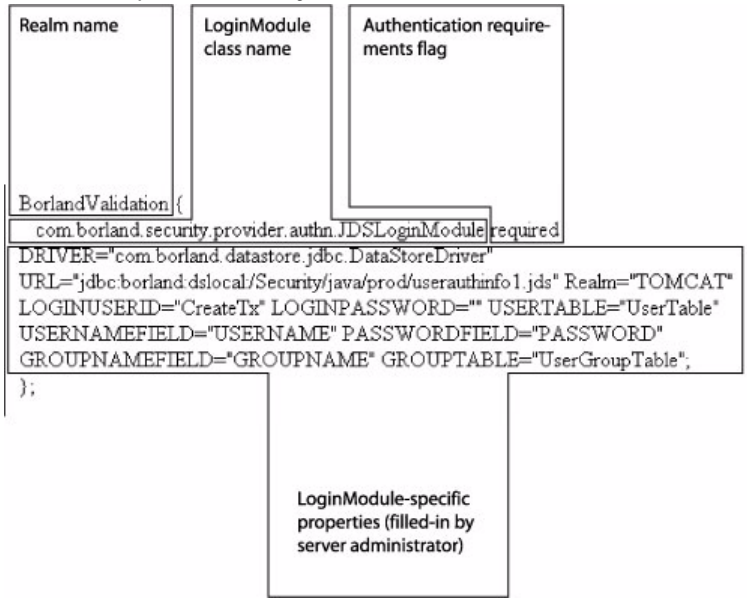
Associating a LoginModule with a realm

The VisiSecure uses the JAAS configuration file to associate a LoginModule with a realm and store that information. The JAAS configuration file contains an entry for each authentication realm. The following is an example of a JAAS configuration entry:

```
MyLDAPRealm {
 com.borland.security.provider.authn.LDAPModule required URL=ldap://
directory.borland.com:389
}
```

The following figure shows the elements of a realm entry in the JAAS configuration file.

Figure 2.1 Realm entry in a JAAS config



A server can support multiple realms. This allows clients to authenticate to any one of those realms. In order for a server to support multiple realms, all you need to do is configure the server with that many configuration entries. The name of the configuration entries is not predefined and can be user defined, for example `PayrollDatabase`.

Note

There must be at least one LoginModule with the authentication requirements flag`=required`.

Syntax of a realm entry

The following code sample shows the generic syntax for a realm entry:

```
//server-side realms for clients to authenticate against
realm-name {
   loginModule-class-name required|sufficient|requisite|optional
   [loginModule-properties];
   ...
};
```

Note

The semicolon (;) character serves as the end-of-line for each LoginModule entry.

The following four elements are found in the realm entry:

- **Realm Name:** The logical name of the authentication realm represented by the corresponding LoginModule configuration
- **LoginModule Name:** The fully-qualified class name of the LoginModule to be used

- **Authentication Requirements Flag:** There are four values for this flag - `required`, `requisite`, `sufficient`, and `optional`.
  You must provide a flag value for each LoginModule in the realm entry. Overall authentication succeeds only if all `required` and `requisite` LoginModules succeed. If a `sufficient` LoginModule is configured and succeeds, then only the `required` and `requisite` LoginModules listed prior to that `sufficient` LoginModule need to have succeeded for the overall authentication to succeed. If no `required` or `requisite` LoginModules are configured for an application, then at least one `sufficient` or `optional` LoginModule must succeed. The four flag values are defined as follows:
  - **required**: The LoginModule is required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list for each realm.
  - **requisite**: The LoginModule is required to succeed. If it succeeds, authentication continues down the LoginModule list in the realm entry. If it fails, control immediately returns to the application—that is, authentication does not proceed down the LoginModule list.
  - **sufficient**: The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application—again, authentication does not proceed down the LoginModule list. If it fails, authentication continues down the list.
  - **optional**: The LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.
- **LoginModule-specific properties**:Each LoginModule may have properties that need to be provided by the server administrator. The necessary properties for each LoginModule provided by Borland are described below.

Borland LoginModules

Borland provides the following common LoginModules for server and client authentication. These LoginModules are used for both client authentication and authentication of the VisiSecure server itself to its operating environment.

Not all LoginModules have the same properties, and your own LoginModules may have different properties as well. Each LoginModule included with VisiBroker is described below, its syntax and properties explained, and a realm entry code sample is provided.

- "Basic LoginModule" -This LoginModule uses a proprietary schema to store and retrieve user information. It uses standard JDBC to store its data in any relational database. This module also supports the proprietary schema used by the Tomcat JDBC realm.
- "JDBC LoginModule" - This LoginModule uses a standard JDBC database interface to authenticate the user against native database user tables.
- "LDAP LoginModule" - This LoginModule is similar to the JDBC LoginModule, but uses LDAP as its authentication back-end.
- "Host LoginModule" - This LoginModule is used for authenticating the operating system hosting the server. **This is the only LoginModule supported for C++.**

For creating the GUI Login, you will have to set the following property:
`com.borland.security.provider.authn.DialogCallbackHandler`

Using a Vault

When running clients, the security subsystem has the opportunity to interact with users to acquire credentials for authentication. This is done using a callback handler as

defined by JAAS. However, when running servers (your Visibroker server or a Partition), it is not desirable or even possible to have user interaction at start up time. A typical example of this is when the server is started as a service at the startup time of a host or from a automated script of some sort.

The *vault* is designed to provide the identity information to the security subsystem in such environments. Vault is merely a tool to replace the user interaction. It is not directly tied to the security subsystem.

In other words, a vault does not contain authenticated credentials. The security service will perform all appropriate authentication, but will receive information from the vault rather than by interacting with a callback handler. Due to the fact that no user interaction is required, the data in the vault, while sufficiently secure, does contain sensitive information (the usernames and passwords). Hence the vault file that is used for authentication of such servers must be protected using host security mechanisms (file permissions for example) or other equivalent approaches.

Creating a Vault

To create a vault, you can use the `vaultgen` command-line tool from your installation's `bin` directory. It's usage is as follows:

```
vaultgen [<driver-options>] -config <config.jaas-file> -vault <vault-name> [<options>]
<command>
```

`<driver-options>` are optional, and can be any of the following:

- `-J<option>`: passes a `-J` Java option directly to the JVM
- `-VBJVersion`: prints VBJ version information
- `-VBJDebug`: prints VBJ debugging information
- `-VBJClasspath`: specify a classpath that will precede the `CLASSPATH` environment variable
- `-VBJProp <name=value>`: passes the name/value pair to the VM
- `-VBJjavavm`: specify the path to the Java VM
- `-VBJaddJar <jar-file>`: appends the JAR file to the `CLASSPATH` before executing the VM

`-config <config.jaas-file>` points to the location of the `config.jaas` file containing the realms, that the identities in the vault will authenticate to.

`-vault <vault-name>` is the path to the vault to be generated. You must also specify an existing vault in order to add additional identities to it.

`<options>` are other optional arguments, and can be any of the following:

- `-?, -h, -help, -usage`: prints usage information
- `-driverusage`: prints usage information, including driver options
- `-interactive`: enables an interactive shell

`<command>` is the command you want `vaultgen` to execute. You can select any one of the following:

- `login <realm|formatted-target>`: establishes an identity in the vault for a given realm or formatted target. The identity is first established when the vault is used for login during system startup.
- `logout <realm|formatted-target>`: removes an identity from the vault for a given realm or formatted target.
- `runas <alias> <realm>`: configures a run-as alias with the identity provided for a given realm.
- `removealias <alias>`: removes a configured run-as alias from the vault.
- `realms`: lists the available realms for this configuration.
- `mechanisms`: lists the available mechanisms (for formatted targets) for this configuration.

- `aliases`: lists configured aliases in the vault.
- `identities`: lists configured identities in the vault.

Example - using VaultGen

Let's look at an example of VaultGen. Let's say we want to create a vault called `MyVault` for use with a domain called `base`. First, we need to know which security profile the domain is using so that we can reference its `config.jaas` file. We check the value of the domain's `vbroker.security.profile` property in the domain's `orb.properties` file:

```
#
# Security for the user domain
#
# Disable user domain security by default
vbroker.security.profile=default
vbroker.security.vault=${properties.file.path}/../security/scu_vault
```

The name of the security profile is `default`. This tells us that the path to the profile's `config.jaas` file is:

Now we can check which realms are contained in the profile for which we want to create identities. We navigate to the installation's `bin` directory, and use the `realms` command:

```
prompt> vaultgen -config config.jaas -vault myVault realms
```

`vaultgen` tells us the following realms are available:

```
The following realms are available:
- UserRealm
- MikeRealm
- BenRealm
```

Next we execute `vaultgen` using the `login` command:

```
prompt> vaultgen -config config.jaas -vault myVault login UserRealm
```

`vaultgen` prompts us for the username and password for the `UserRealm`, which we enter. We then repeat the process for each additional realm. At the end of each command, `vaultgen` informs us that it has logged-in the new identity and saved changes to `MyRealm`.

```
Logged into realm BenRealm
Generating Vault to MyVault
```

The vault is created in the directory you specify in the command, in this case the `bin` directory. A good place to put the actual vault files are in the domain's `security` directory, located in:

```
<install-dir>/var/domains/<domain-name>/adm/security/
```

Example: Using API

This example illustrates the use of Security Context Interface's APIs `generateVault(std::osstream& os, CSI::UTF8String& pass)`, `login(std::istream& is)`) which can be used to explicitly login to the server. The example given here has all the basic setting needed to secure an application.

The API generateVault will take a file output stream and stores the userid/password and realm in a file. It generates a byte stream from the login information by encrypting the login (Userid/password/realm) information. After encrypting, it closes the files and logs out.

During authorization, the system uses the file created above to login rather than asking the user to provide the information using FileInputStream API and gets the security Context from the ORB and logs in using the file.

The example also illustrates the use of VisiBroker properties and JAAS configuration file to secure your application. The example client and server uses username/password authentication of client on the server and also for server's self authentication.

Look at the different properties files: server.properties, client.properties and Config files: server.config and client config for clients in the basic vault example is in the \\ Borland\VisiBroker\examples\vbroker\security\basic folder. The Bank Agent example has a simple AccountManager interface to open an Account and to query the balance in that account.

To run the example, first generate the vault file as given below.

To generate vault file:

In the command prompt, enter the following command in the server window:

```
prompt% Server -DORBpropStorage=cpp_server.properties -genVault <vaultfileName>
```

It will ask for the userid/password, enter Host Login Name and password for the current system. This information gets stored in the vaultfile.

To run the server with out providing authentication information:

```
prompt% Server -DORBpropStorage=cpp_server.properties -useVault <vaultfileName>
```

(start Server -DORBpropStorage=cpp_server.properties -useVault <vaultfileName> on Windows)

To run the client, simply use the command:

```
prompt% Client -DORBpropStorage=cpp_client.properties
-Dvbroker.orb.dynamicLibs=<vbsec library>
```

where: <vbsec library>

(in windows): `vbsec.dll located in %VBROKERDIR%/bin directory`

(in UNIX): `libvbsec shared library located in $VBROKERDIR/lib`

The vaultfile uses this file information to log the user without user interaction.

It will ask for the userid/password, enter the Host Login Name and password for the current system

Certificate Revocation List

For C++ only

When signed public key certificates are created by a Certificate Authority (CA), each certificate has an expiry date that indicates when it is no longer valid. However, in order to address the case where a certificate becomes invalid for some reason before the date of expiry, the Certificate Revocation List (CRL) feature is provided for VisiSecure for C++. For more information about Certificate Authorities (CA)s, see the “Certificates and Certificate Authority”.

Using the VisiSecure for C++ Certificate Revocation List (CRL) feature, you can set up CRLs and check peer certificates against this list during SSL handshake communication.

The CRL files can be in either DER binary format or base 64 text format. When an application adds a trusted certificate into a VBSSLContext instance, the coresponding CRL of that trusted certificate can be passed as second parameter of the call to the addTrustedCertificate() method. For this, the physical CRL bytes (if in DER) or string characters (if in B64) need to be wrapped in an instance of CORBAsec::ASN1Object, which is actually a CORBA octet sequence. Please see VBSSLContext API in header file vbssp.h.

```
class _VBSECEXPORT VBSSLContext
{
  ...
  virtual void
  addTrustedCertificate( const CORBAsec::X509Cert& trusted,
                                  const CORBAsec::ASN1Object* crl = NULL ) = 0;
  ...
};
```

Multiple trusted certificates can be installed along with their respective CRLs by means of multiple calls on a VBSSLContext instance. Concrete examples of CRL installation is in the bank_ssl example.

The method addTrustedCertificate() involves cryptographic verification to make sure that the CRL is signed using the private key of the public key in the certificate.

Applications can call addTrustedCertificate() with only the first parameter, in which case it is assumed that the trusted certificate has no coresponding CRL.

Note

There can be more than one CRL file within the CRL Repository directory structure.

Once the CRLs are loaded, VisiSecure examines all certificates sent by a peer during SSL handshake. If any of the peer certificates appears in the CRLs, an exception will be thrown and the connection will be refused.

3

Authorization

Authorization is the process of verifying that the user has the authority to perform the requested operations on the server. For example, when a client accesses an enterprise bean method' the application server must verify that the user of the client has the authority to perform such an access. Authorization occurs after authentication (confirming the user's identity).

Access Control

Authorization occurs after the user proves who he or she is (Authentication). Authorization is the process of making access control decisions on requested resources for an authenticated entity based on certain security attributes or privileges. Following Java Security Architecture, VisiBroker adopts the notion of *permission* in authorization. In VisiSecure, resource authorization decisions are based on permissions. Borland uses a proprietary authorization framework based on users and roles to accomplish authorization. For example, when a client accesses a CORBA or Web request enterprise bean method, the application server must verify that the user of the client has the authority to perform such an access. This process is called access control or authorization.

Access Control List

Authorization is based on the user's identity and an *access control list* (ACL), which is a list roles. Typically, an access control list specifies a set of roles that can use a particular resource. It also designates the set of people whose attributes match those of particular roles, and thus are allowed to perform those roles.

Roles-based access control

ACL in Corba is provided programatically The VisiSecure uses a role database (a file whose default name is `roles.db`) to associate user identities with roles. If a user is associated with at least one allowed role, the user may access the method. For more information, refer to “Configuring authorization using the rolemap file”.

Pluggable Authorization

VisiSecure provides the ability to plug-in an authorization service that can map users to roles. The implementer of the Authorization Service provides the collection of permission objects granted access to certain resources.

A new class, `RolePermission` is defined to represent “role” as permission. The RolePermission class provides the basis for authorization and trust in the VisiSecure system. The Authorization Services Provider in turn provides the implementation on the homogeneous collection of `RolePermissions` contained for an association between given privileges and a particular resource.

For more information on RolePermission class, refer to “vbsec::RolePermission”

The Authorization service is tightly connected with the concept of Authorization domain—each domain has exactly one Authorization Services Provider implementation. The Authorization domain is the bridge between VisiSecure system and the authorization service implementation.

During the initialization of the ORB itself, the authorization domains defined by the property `vbroker.security.authDomains` are constructed, while the Authorization Services Provider implementation is instantiated during the construction of the Authorization domain itself.

The Authorization domain defines the set of rules that determine whether a user belongs to a logical “role” or not.

The implementer of the Authorization Service provides the collection of permission objects granted access to certain resources. Whenever an access decision is going to be made, the AuthorizationServicesProvider is consulted. The Authorization Service is closely associated with the Authorization domain concept. An Authorization Service is installed per each Authorization domain implementation, and functions only for that particular Authorization domain.

The `AuthorizationServicesProvider` is initialized during the construction of its corresponding Authorization domain. Use the following property to set the implementing class for the AuthorizationServicesProvider:

```
vbroker.security.domain.<domain-name>.provider
```

During runtime, this property is loaded by way of Java reflection.

Another import functionality of the Authorization Service is to return the run-as alias for a particular role given. The security service is configured with a set of identities, identified by aliases. When resources request to “run-as” a given role the AuthorizationServices again is consulted to return the alias that must be used to “run-as” in the context of the rules specified for this authorization domain.

For more information on RolePermission class, refer to “vbsec::AuthorizationServicesProvider”.

Configuring authorization using the rolemap file

You can configure authorization by creating your own Authorization Rolemap by hand.

What is a rolemap

The authorization rolemap is captured in a `.rolemap` file. Typically, you would name this file after your authorization domain. The rolemap file, also called *Role DB*, is a map of users to roles. The rolemap designates the set of people whose attributes match those of particular roles, and who are then allowed to perform those roles.

VisiSecure provides a mechanism for specifying role names and a set of attributes which define the role.

Syntax of Role DB

The Role DB file itself has the following form, and can contain multiple role entries:

```
role-name {
   assertion1 [, assertion2, ... ]
   ...
   [assertion-n]
   ...
}
```

```
role-name2 {
   assertion3 [, assertion4, ... ]
   ...
   [assertion-n]
   ...
}
```

A role entry is made up of a role name and a list of rules within curly braces (“{}”). A role must be made up of one or more rules. Each *rule* is a single line containing a list of comma-separated *assertions* for proper access identifications. Similarly, each rule must contain one or more assertions.

Each line in the Role Entry is a *rule*. Rules are read top-to-bottom, and authorization proceed until one succeeds or none succeed. That is, each rule is read as though separated by an “OR” operator. *Assertions* are separated on the same line by a comma (“,”). Assertions are read left-to-right, and all assertions must succeed in order for the rule to succeed. That is, each assertions in a rule is read as though separated by an “AND” operator.

Each rule must contain all necessary security information for a given Principal's security credentials. That is, each principal must have at least those attributes required from the rule—or exactly all the listed attributes. Otherwise authorization will not succeed. For more information on specifying rules, see “Specifying rules for authorization”.

For example, the contents of Role DB could be:

```
ServerAdministrator {
  CN=*, OU=Security, O=Borland, L=San Mateo, S=California, C=US
  *(CN=admin)
  *(GROUP=administrators)
}

Customer {
  role=ServerAdministrator
  *(CN=borland)
  *(CN=pclare)
  *(CN=jeeves)
  *(GROUP=RegularUsers)
}
```

This defines two roles, `ServerAdministrator` and `Customer` along with a set of rules and attributes which define them.

Once the rolemap file is complete, it can be referenced using properties.

Defining Roles in Roles DB

Role DB is a text file containing the roles and the rules associated with those roles. Each role in Role DB constitutes a *role entry*.

In VisiBroker, the location of the rolemap file is specified using the property:
`vbroker.security.domain.<authorization-domain>.rolemap_path`

The Role DB file is used to determine the access rights of principals (client identities). Each role defined in the Role DB has client identities assigned to it. Access rights are granted based on roles rather than specific client identities.

For example:

The application may recognize a Sales Clerk role. User identities for all sales clerks can be assigned to the Sales Clerk role. Later, the Sales Clerk role is granted the right to perform certain operations, such as an `add_purchase_order` method, for example. All sales clerks associated with the Sales Clerk role are able to perform `add_purchase_order`.

Modifying authorization rolemap file

You can modify the authorization rolemap files by editing the rolemap file using properties given in the example directory. You can specify rolenames and attributes and thus associate users with roles. A role must be made up of one or more rules. For more information on rules and role entries, refer to “Specifying rules for authorization.”.

For configuring database to store users, credentials and attibutes, refer to “Creating groups and associating users with groups.”

Specifying rules for authorization

Assertion syntax

There are a variety of ways to specify rules using logical operators with attribute/value pairs that represent the access identifications necessary for authorization. There is also a simplified syntax using the wildcard character (“*”) to give your rules more flexibility. Both of these are discussed below.

Using logical operators with assertions

Two logical operators are available in specifying attribute/value pairs.

Operator	Description	Example
attribute **=** value	**equals**: attribute must equal value for authorization rule to succeed.	`CN=Russ Simmons`
attribute **!=** value	**not equal**: attribute must not equal value for authorization rule to succeed.	`CN!=Rick Farber`

A *value* can be any string, but the wildcard character, “*” has special uses. For example, the attribute/value pair `GROUP=*` matches for all GROUPs. The following role has two associated rules:

```
manager {
   CN=Kitty, GROUP=*
   GROUP=SalesForce1, CN=*
}
```

The role manager has two rules associated with it. In the first rule, anyone named `Kitty` is authorized for `manager`, regardless of Kitty's associated group at the time. The second rule authorizes anyone in the group `SalesForce1`, regardless of their common-name (`CN`).

Wildcard assertions

For complicated security hierarchies, you must be cautious to look for only one or two attributes from the hierarchy in order to authorize a principal. Borland's security hierarchy starts with GROUPs at the top, then branches out into ORGANIZATIONs (O) and ORGANIZATIONAL UNITS (OU), and finally settles on COMMON NAMEs (CN).

For example, you may want to define a security role called SalesSupervisor that allows method permissions for managers of the sales force. (For this example, “sales” is the ORGANIZATION and “managers” is the ORGANIZATIONAL UNIT. You could do so with the following rule:

```
SalesSupervisor {
   GROUP=*, O=sales, OU=managers, CN=*
}
```

This rule does not specify values for GROUP or for COMMON NAME (presumably because they are not necessary). But remember, each rule must represent all possible values for a Principal's credentials. There are other means of representing this same information in a smaller space using *wildcard assertions*.

You make a wildcard assertion by placing the wildcard character (“*”) in front of the assertion(s) in one of the two ways. You may place the wildcard character in front of a single assertion, meaning that all possible security attributes are accepted but they

must contain the single assertion. Or, you may place the wildcard character in front of a list of assertions separated by commas within parentheses. This means all possible security attributes are accepted but they *must* contain the assertions listed in the parentheses.

Making use of wildcard assertions, the role could also look like this:

```
SalesSupervisor {
   *O=sales, *OU=managers
}
```

Or, even more simply:

```
SalesSupervisor {
   *(O=sales, OU=managers)
}
```

All three code samples are different versions of the same rule.

Other assertions

Each role provides limited extensibility to others. You may, as a part of a role entry, specify a `role=existing-role-name` assertion that can extend an earlier role. You can also use customized code as your authorization mechanism rather than Role DB syntax by using the Authorization Provider Interface.

Recycling an existing role

You can refer to the rules from an existing role by using the rule-reference assertion—`role=role-name`.

For example, let's say we have a group of marketers who are also sales supervisors that need to be authorized to the same code as Sales Supervisors. Building upon the `SalesSupervisor` code sample, we can create a new role entry as follows:

```
MarketSales {
   role=SalesSupervisor
   *(OU=marketing)
}
```

Now, everyone in role SalesSupervisor has access to the MarketSales role, as does anyone in the “marketing” OU.

Configuring authorization domains

The authorization domain defines the set of rules that determine whether a user belongs to a logical “role” or not.

The authorization domain is the bridge between VisiSecure system and the authorization service implementation.

During the initialization of the ORB itself, the authorization domain is defined by the property `vbroker.security.authDomains`.

Specifying names to authorization domain

Each Role DB file is associated with an *authorization domain*. An authorization domain is a security context that is used to separate role DBs and hence their authorization permissions. For more information on the authorization domain in the context of the basic security model, see “Basic security model”.

You may use as many authorization domains as you wish, provided they are all registered with the VisiBroker ORB. You must do the following for each of your authorization domains:

- give it a name,
- set up default access,

– set up the Role DB,

To accomplish these items, the following ORB properties must be set. For more information about these properties, see "Security Properties for Java" or "Security Properties for C++":

Naming authorization domains

You can give the authorization domain a name and list them using the property:

```
vbroker.security.authDomains=<domain1> [, <domain2>, <domain3>, ...]
```

Setting up default access

You can set up the default access and decide whether or not to grant access to the domain in the absence of security roles for <domain-name>.

The property used to set up the default access is

```
vbroker.security.domain.<domain-name>.defaultAccessRole=grant|deny
```

Setting up RoleDB

Path of the Role DB file is associated with the authorization domain `domain-name`. Although this can be a relative path, Borland recommends you make this path fully-qualified.

The property you use to set up the RoleDB is

```
vbroker.security.domain.<domain-name>.rolemap_path=<path>
```

Configuring authorization domains to run-as alias

Authorization domains are then configured to run-as a given alias for a role in that domain. When a request is made to run-as a given role, then the authorization domain for that context is consulted to get the corresponding run-as alias. The run-as map is then consulted to get the identity corresponding to that alias, and this identity is used.

Run-as identities can also be configured to be certificate identities and not just username/password identities.

Run-as Alias is useful in particular when there are clients, middle-tier servers and end-tier servers.

To set up run-as alias on corba application level:

1. Set the property in server.propeties file. This property specifies the name of the run-as role. The value can be either use-caller-identity to have the caller principal be in the run-as role, or specify an alias for a run-as principal for the run-as role name
   ```
   vbroker.security.domain.<domain name>.runas.<run_as_role_name>
   ```
2. Specify a list of trusted roles as specified in the authorization domain. This is uniquely identified for each trust assertion rule as a list of digits.
3. Trust all the assertion made by peers by setting the property below to true.
   ```
   vbroker.security.assertions.trust.all
   ```

Run-as Alias

A Run-as Alias is a string identifying an authentication identity. It is defined in the vault and scoped within the VisiBroker ORB. This alias then represents a particular user. The identity is mapped to the alias using either the Context APIs or by defining it in the vault. The vault can contain a list of run-as aliases and the corresponding authenticating credentials for the identity to run-as. In both cases, the authenticating credentials (from the vault or wallet) are passed to the LoginModules, which authenticate those credentials and set them as fully authenticated identities corresponding to those credentials in the run-as map.

You can set up an identity for the run-as role <role-name>. The alias denotes an alias in the vault.

Use `use-caller-identity` to use the caller principal itself as the principal identity for the run-as role.

The propery you use to set up aliases is

`vbroker.security.domain.<domain-name>.runas.<role-name>=<alias>|use-caller-identity`

Note

Run-as aliases are not available under C++.

Creating Vault

To create a new vault, modify the following files below in the corbaauthz example in the \\VisiBroker\examples\vbroker\security\example folder:

- java_server.properties
- AccountImpl.java
- Server.java
- Bank.idl

You may need to add the following files:

- ConverterImpl.java
- ConverterServer.java (many parts are just cut-paste Server.java)
- Create a new vault named 'fault' as shown in created in step (4)

In the java server properties file, add the following properties:

```
vbroker.security.domain.bank.runas.jeeves_runasrole=jeeves_alias
vbroker.security.assertions.trust.all=true
```

In Bank.idl file, make the following changes

```
module Bank {
  interface Converter {
    float toSGD( in float USD);
  };
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

In AccountImpl.java file, modify as shown in bold below.

```
// AccountImpl.java
import com.borland.security.csiv2.ObjectAccessPolicy;
import org.omg.CORBA.BAD_OPERATION;
import org.omg.CORBA.ORB;

public class AccountImpl extends Bank.AccountPOA implements ObjectAccessPolicy {

  public String[] getRequiredRoles(String op) {
    if (op.equals("balance")) {
      return new String[] {"Customer", "Teller"};
    }
    throw new BAD_OPERATION("No operation named " + op);
  }
```

```
  public String getRunAsRole(String op) {
//    return "jeeves_runasrole";
    return null;
  }

  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _converter.toSGD( _balance );
  }
  private float _balance;


  public static Bank.Converter _converter;
}
```

In server.java, to write the AccountManager's reference to a file for the client to access, add the following commands:

```
   FileWriter output = new FileWriter("bank.ior");
      output.write(orb.object_to_string(object));
      output.close();
      System.out.println(object + " is ready.");
      // Wait for incoming requests
      orb.run();
    }


// AccountManagerImpl.java
import org.omg.PortableServer.*;
import com.borland.security.csiv2.ObjectAccessPolicy;
import org.omg.CORBA.BAD_OPERATION;
import org.omg.CORBA.ORB;

import java.util.*;

public class ConverterImpl extends Bank.ConverterPOA implements ObjectAccessPolicy {

  public String[] getRequiredRoles(String op) {
    if (op.equals("toSGD")) {
      return new String[] {"Manager"};
    }
    throw new BAD_OPERATION("No operation named " + op);
  }

  public String getRunAsRole(String op) {
    return null;
  }

  public float toSGD( float USD) {
    System.out.println( "*** Converter.toSGD is called");
    javax.security.auth.Subject caller = _current.getCallerSubject();
    System.out.println( "THE CALLER = " + caller.toString() );
    return _rate * USD;
  }

  // Use const for now
  private static final float _rate = 1.65676f;
  com.borland.security.Current _current;

  ConverterImpl( com.borland.security.Current current) {
    _current = current;
  }

}
```

Add a new file ConverterServer.java. This file is the same server.java file that is in the corba authz example. Add the following that is given in bold:

```
// Server.java
import org.omg.PortableServer.*;
import java.io.*;
import org.omg.CORBA.Any;
import org.omg.CORBA.Policy;
import org.omg.CORBA.PolicyManager;
import org.omg.CORBA.PolicyManagerHelper;
import org.omg.CORBA.SetOverrideType;
import com.borland.security.csiv2.SERVER_QOP_CONFIG_TYPE;
import com.borland.security.csiv2.ServerQoPConfigDefaultFactory;
import com.borland.security.csiv2.ServerQoPConfig;
import com.borland.security.csiv2.ServerQoPConfigHelper;
import com.borland.security.csiv2.ServerQoPPolicy;
import com.borland.security.csiv2.AccessPolicyManager;
import com.borland.security.csiv2.ObjectAccessPolicy;

public class ConverterServer {

  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // get a reference to the root POA
      POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      ServerQoPConfig config =
        new ServerQoPConfigDefaultFactory().create(false,
                                                   ServerQoPPolicy.ALL,
                                                   true,
                                                   new AccessPolicyManager() {
          public String domain () {
            return "bank";
          }

          public ObjectAccessPolicy getAccessPolicy (Servant servant,
                                                     byte[] id,
                                                     byte[] adapter_id) {
            return (ObjectAccessPolicy) servant;
          }
        });
      Any any = orb.create_any();
      ServerQoPConfigHelper.insert(any, config);
      Policy qop = orb.create_policy(SERVER_QOP_CONFIG_TYPE.value, any);

      PolicyManager polmgr =
        PolicyManagerHelper.narrow(orb.resolve_initial_references
("ORBPolicyManager"));
      polmgr.set_policy_overrides(new Policy[] {qop},SetOverrideType.SET_OVERRIDE);

      // Create policies for our persistent POA
      org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
      };
      // Create myPOA with the right policies
      POA myPOA = rootPOA.create_POA( "converter_poa", rootPOA.the_POAManager(),
                                      policies );
      // Create the servant
      com.borland.security.Current current =
          (com.borland.security.Current) orb.resolve_initial_references (
            "VBSecurityCurrent"
          );
      ConverterImpl converterServant = new ConverterImpl(current);
```

```
        // Decide on the ID for the servant
        byte[] converterId = "CurrencyConverter".getBytes();

        // Activate the servant with the ID on myPOA
        myPOA.activate_object_with_id( converterId, converterServant);

        // Activate the POA manager
        rootPOA.the_POAManager().activate();

        // convert servant to an object reference
        org.omg.CORBA.Object object = myPOA.servant_to_reference(converterServant);

        // Write the AccountManager's reference to a file,
        // so clients can access it.
        FileWriter output = new FileWriter("converter.ior");
        output.write(orb.object_to_string(object));
        output.close();
        System.out.println(object + " is ready.");
        // Wait for incoming requests
        orb.run();
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }

  }
```

In this example, we are creating a vault called fault:

1. At the command prompt, enter the following command
   `> vaultgen -vault fault -config java_server.config -interactive`
2. Enter "`quit`" to quit.
3. Login to the realm
   `> login myrealm`

```
JDataStore:  Developer's License (no connection limit)
JDataStore:  Copyright (c) 1996-2004 Borland Software Corporation.  All rights
reserved.
JDataStore:  License for JDataStore development only - not for redistribution
JDataStore:  Registered to:
JDataStore:  JDataStore
JDataStore:  Developer's license with unlimited connections
```

4. Enter authentication information for realm `myrealm`
5. Enter username: `admin`
6. Enter password: `admin`

   Logged into realm myrealm
7. > runas `jeeves_alias myrealm`
8. Enter authentication information for realm `myrealm`
9. Enter username: `jeeves`
10. Enter password: `jeeves`

    Added runas alias jeeves_alias
11. > quit

    Generating Vault to fault

To launch run-as alias:

Open three different commandline consoles (DOS prompts on NT)

Start the osagent

1. In console1, launch client as follows
   ```
   vbj -DORBpropStorage=java_client.properties Client
   ```
2. When prompted, enter borland/borland.
3. In console2, launch server as follows
   ```
   vbj -DORBpropStorage=java_server.properties Server
   ```
4. In console 3, launch currency converter server as follows
   ```
   vbj -DORBpropStorage=java_server.properties ConverterServer
   ```
5. In console 3,
   ```
   *** Converter.toSGD is called
   THE CALLER = Subject:
   Principal: jeeves@myrealm
   Public Credential: Privileges for jeeves@myrealm
   Private Credential: Destroyed authentication context for null
   Private Credential: com.borland.security.provider.ATSCodec$EncoderCache@e93999
   ```
6. Open and edit the AccountImpl.java file and change the following:
   ```
            public String getRunAsRole(String op) {
              return "jeeves_runasrole";
            }
   to
   public String getRunAsRole(String op) {
              return null;
            }
   ```
7. In console1, launch client as follows
   ```
   vbj -DORBpropStorage=java_client.properties Client
   ```
8. When prompted, login as borland/borland

See console3 again,

```
*** Converter.toSGD is called
THE CALLER = Subject:
Principal: borland@myrealm
Public Credential: Privileges for borland@myrealm
Private Credential: Destroyed authentication context for null
Private Credential: com.borland.security.provider.ATSCodec$EncoderCache@e9399
9
```

This time, the original caller borland@myrealm is propagated properly all the way to the ConverterServer. But previously, this identity was translated into `jeeves@myrealm`

There are three things that together specifies `jeeves@myrealm`

1. `AccountImpl.java getRunAsRole()` returns "`jeeves_runasrole`"
2. java_server.properties translates "`jeeves_runasrole`" into "`jeeves_alias`" using

   ```
   vbroker.security.domain.bank.runas.jeeves_runasrole=jeeves_alias
   ```

3. The vault contains runas entry with alias "`jeeves_alias`".

Run-as mapping

Note

Run-as mapping is not available under C++.

Setting the `vbroker.security.domain.<domain-name>.runas.<role-name>` property effectively maps an alias to a bean's run-as role. Upon successful authorization, but before method invocation, the container checks the Run-as role specified in the EJB's deployment descriptor for the called method. If a run-as role exists, the container checks to see if there is an alias as well. If there is, when the bean makes an outgoing invocation it switches to the identity for that alias.

If, however, no alias is specified (that is, the run-as role name is set to `use-caller-identity`), the caller principal name is used.

Setting up authorization for CORBA objects

Authorization in the CORBA environment allows only identities in specific roles for a given object that can access that object. An object's access policy is specified by means of a Quality of Protection policy for the Portable Object Adapter (POA) hosting the object in question. Note that access policies can only be applied at the POA level.

Rolemaps are also used to implement authorization for CORBA objects.

In order to set up authorization for an object, you need to perform the following:

1. Create a `ServerQopPolicy`.
2. Initialize the `ServerQopPolicy` with a `ServerQopConfig` object.
3. Implement an `AccessPolicyManager` interface, which takes the following form:

Java

```
interface AccessPolicyManager {
   public java.lang.String domain();
   public com.borland.security.csiv2.ObjectAccessPolicy getAccessPolicy(
      org.omg.PortableServer.Servant servant, byte[] object_id byte [] adapter_id);
}
```

C++

```
class AccessPolicyManager {
    public:
        virtual char* domain() =0;
        ObjectAccessPolicy_ptr getAccessPolicy(PortableServer_ServantBase* _servant,
        const ::PortableServer::ObjectId& id,
        const::CORBA::OctetSequence& _adapter_id) =0;
}
```

Setting up name

This interface should return the authorization domain from the `domain()` method and set the access manager in the `ServerQopConfig` object. The domain specifies the name of the authorization domain associated with the proper rolemap.

You set the location and name of the rolemap by setting the property:

```
vbroker.security.domain.<authorization-domain-name>.<rolemap-path>
```

where `<authorization-domain-name>` is a tautology, and `<rolemap-path>` is a relative path to the rolemap file.

Setting up default access

The `getAccessPolicy()` method takes an instance of the servant, the object identity, and the adapter identity and returns an implementation of the `ObjectAccessPolicy` interface.

1. It implements the `ObjectAccessPolicy` interface that returns the required roles and a run-as role for accessing a method of the object. There is no difference between J2EE and CORBA run-as roles in Borland's implementation. The `ObjectAccessPolicy` interface takes the following form:

Java

```
interface ObjectAccessPolicy {
    public java.lang.String[] getRequiredRoles(java.lang.String method);
    public java.lang,String getRunAsRole(java.lang.String method);
}
```

C++

```
class ObjectAccessPolicy {
    public:
        getRequiredRoles (const char* _method) =0;
}
```

Setting up Alias(es)

The `getRequiredRoles()` method takes a method name as its argument and returns a sequence of roles. The `getRunAsRole()` method returns a run-as role, if any, for accessing the method.

Identities can be supplied using Callback Handlers. For more details, see "Authentication."

Configuring authorization requirements

You must configure authorization requirement for the components in the server, as the client needs to have these authorizations in order to access these components in the server

In the corbaauthz example in \\VisiBroker\examples\vbroker\security folder, the authorization requirement for the BankManager object is that the clients should be a member of the "Manager" role and for the Account it is "Customer" or "Teller" role.

The rolemap file contains the authorization data from the Role DB file
Members of the roles Manager, Customer and Teller are described in the bank.rolemap file whose snippet is shown below:

Example for bank.rolemap file for Java:

```
Manager {
   *CN=admin
   *group=user
}
Customer {
   *CN=admin
}
Teller {
   *CN=admin
   *group=user
}
```

Example for bank.rolemap file for C++:

```
Manager {
   *group=cceng
}
Customer {
   *group=cceng
}
Teller {
   *group=cceng
}
```

Any authenticated user with username=Administrator is a member of role 'Manager'. Any authenticated user with group=cceng is a member of both role 'Customer' and role 'Teller'.

You can use this example and change the username and group to use valid, existing username and group in your system as required.

The example illustrates the use of VisiBroker properties and JAAS configuration file to secure your application. The example client and server uses username/password authentication of client on the server and also for server's self authentication.

Look at the different properties files (server.properties, client.properties) and config files (server.config and client.config) in the \\Borland\VisiBroker\examples\vbroker\security/corbaauthz folder.

The server or the client configuration file is the JAAS configuration file which defines the login modules.

To enable security, you must set up certain properties. Following are the properties set up in the server or client properties file:

Properties	Description
vbroker.security.disable=false	The default value is false. If set to true, disables all security services.
vbroker.security.login=true	If this property is set to true, during initialization, this property tries to log on to all the realms listed by property vbroker.security.login.realms.
vbroker.security.authentication.config=cpp_server.config	This specifies the path to the configuration file used for authentication. The default value is null.
vbroker.security.authDomains=bank	Specifies a comma-separated list of available authorization domains. For example: vbroker.security.authDomains=domain1,domain2
vbroker.security.domain.bank.rolemap_path=./cpp_bank.rolemap	Specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain <domain_name> specified in: vbroker.security.authDomains.
vbroker.security.domain.bank.defaultAccessRule=grant	Specifies whether to grant or deny access to the domain by default in the absence of security roles for the provided domain. It handles requests for methods not in the rolemap file. Acceptable values are grant or deny.
vbroker.security.peerAuthenticationMode=none	Sets the peer authentication Mode. when set to NONE—Authentication is not required. During handshake, no certificate request will be sent to the peer. Regardless of whether the peer has certificates, a connection will be accepted. There will be no transport identity for the peer. For other authentication mode values, refer to the properties section "vbroker.security.peerAuthenticationMode"
vbroker.security.login.realms=myrealm	This gives a list of comma-separated realms to login to. This is used when login takes place, either through property vbroker.security.login (set to true) or API login.
vbroker.security.authentication.callbackHandler=com.borland.security.provider.authn.HostCallbackHandler	Specifies the callback handler for login modules used for interacting with the user for credentials. You can specify one of the following or your own custom callback handler. For more information, see "VisiSecure for C++ APIs." com.borland.security.provider.authn.CmdLineCallbackHandler com.borland.security.provider.authn.HostCallbackHandler CmdLineCallbackHandler has password echo on, while HostCallbackHandler has password echo off.

The properties allow you to customize the behavior of VisiSecure. Depending on whether your application is Java, C++, or both, you may have to set different properties with different types of values. See “Security Properties for C++” and “Security Properties for Java” for all the properties you can set in this file.

Using vault for a domain

If you are using a vault to store system identities, you associate it with a domain so that it can be used. You do this by setting the domain's `vbroker.security.vault` property in the domain's `orb.properties` file.

Set the property to the location of the domain's vault. For example:

```
vbroker.security.vault=c:/BDP/var/domains/base/adm/security/MyVault
```

Similar to the vault are other properties which only belong to the `orb.properties` file. These include secure listener ports, thread monitoring, and so forth.

As a general rule, add only those properties that can be shared by multiple applications. Otherwise, use the appropriate process-specific ORB properties file to specify the property.

Context Propagation

In addition to ensuring the confidentiality and integrity of transmitted messages, you need to communicate caller identity and authentication information between clients and servers. This is called *delegation*. The caller identity also needs to be maintained in the presence of multiple tiers in an invocation path. This is because a single call to a mid-tier system may result in further calls being invoked on other systems which must be executed based on the privileges attributed to the original caller.

In a distributed environment, it is common for a mid-tier server to make *identity assertions* and act on behalf of the caller. The end-tier server must make a decision whether the assertion is trusted or not. When propagating context, the client transfers the following information:

- **Authentication token**—client's identity and authentication credentials.
- **Identity token**—any *identity assertion* made by this client.
- **Authorization elements**—privilege information that a client may push about the caller and/or itself.

Impersonation

Impersonation is the form of identity assertion where there is no restriction on what resources the mid-tier server can access on the end-tier server. The mid-tier server can perform any task on behalf of the client.

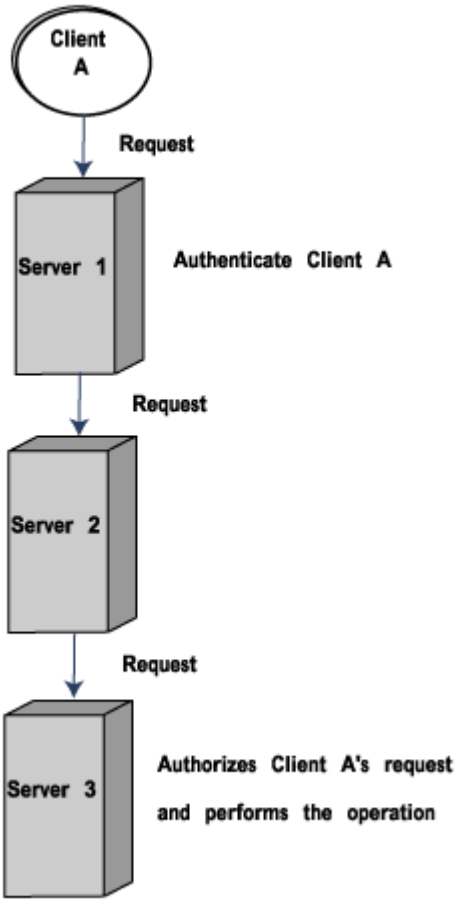
Delegation

Delegation is the form of identity assertion where the client explicitly delegates certain privileges to the server. It is the inverse of impersonation. It is a mechanism to communicate caller identity and authentication information between clients and servers. In this case, the server is allowed to perform only certain actions as dictated by the client. VisiSecure performs only simple delegation.

Identity assertions

Identity assertion occurs when several servers with secure components are involved in a client request. At times, it is necessary for a server to act on behalf of its clients—when a client request is passed from one server to another. This is typical in the case where a client calls a mid-tier server, and the server further needs to call an end-tier server to perform a part of the service requested by the client. At such times, the mid-tier server typically needs to act on behalf of the client. In other words, it needs to let

the end-tier server know that while it (the mid-tier server) is communicating with the end-tier server, access control decisions must be based on the original caller's privileges and not its privileges.



For example, a client request goes to Server 1, and Server 1 performs the authentication of the identity of the client. However, Server 1 passes the client request to Server 2, which may in turn pass the request to Server 3, and so forth.

Each subsequent server (Server 2 and Server 3) can assume that the client identity has been verified by Server 1 and thus the identity is trusted. The server that ultimately fulfills the client request, such as Server 3, need only perform the access control authorization.

By default, the identity is authenticated only at the first tier server and is asserted. It is the asserted identity that propagates to other tiers.

Asserting Identity of the caller

The identity assertion example in the \\VisiBroker\examples\vbroker\security\assertion example folder illustrates the use of identity assertion APIs which can be used to explicitly assert an identity as caller.

This example uses APIs provided by security context to create a new identity and assert the identity as caller before making the invocation. The server first checks if the assertion is made by the trusted peer and then checks if the asserted identity is authorized to make the invocation.

You can change the attribute to suite your own environment before running the example. In this example, server is setup with assertion trust and authorization rules.

To make assertion, use the following command:

```
    for (int i=0; i<argc; ++i) {

      if (strcmp(argv[i], "-assert") == 0) {
        CORBA::Object_var obj1 = orb->resolve_initial_references("VBSecurityContext");
        Context* context = dynamic_cast<Context*>(obj1.in());

        CORBA::Object_var obj2 = orb->resolve_initial_references("VBSecurityCurrent");
        Current* current = dynamic_cast<Current*>(obj2.in());

        CORBA::Object_var obj3 = orb->resolve_initial_references("VBWalletFactory");
        WalletFactory* factory = dynamic_cast<WalletFactory*>(obj3.in());

        Wallet* wallet = factory->createIdentityWallet("asserted", "password",
"myrealm");
        Subject* subject = context->importIdentity(*wallet);
        current->asserting(subject);
      cout << "New caller identity asserted." << endl;
        break;
      }
    }
```

The assertion trust rule on the server requires the asserter, which is the client in this example. The same must be a member of *Asserter* role for authorization domain *bank*.

Members of the role *Asserter* is described in the *bank.rolemap* as follows:

```
Asserter {
*group=cceng
}
```

This means, any authenticated user that belongs to group *cceng* must be a member of role *Asserter*.

The authorization rule on the server requires the caller, which is the identity asserted by the client in this example, must be of *Manager* role for authorization domain *bank.*

Member of the role *Manager* is described in the *bank.rolemap* as follows:

```
Manager {
*cn=asserted
}
```

The client code asserts an identity that user name is *asserted*, thus the asserted identity is member of role *Manager* and access will be granted.

1 Launch the server using the following command:

```
prompt> Server -DORBpropStorage=cpp_server.properties &
(start Server -DORBpropStorage=cpp_server.properties on Windows)
```

2 Launch the client with assertion using the following command:

```
prompt> Client -DORBpropStorage=cpp_client.properties -assert
```

3 Enter your userid/password, when prompted, for the current host machine.

The program runs successfully.

To run the client without assertion:

1 Launch the client using the following command:

```
prompt> Client -DORBpropStorage=cpp_client.properties
```

2 Enter the userid/password, when prompted, for the current host machine.

You shall see exception of `CORBA::NO_PERMISSION` is thrown as only the asserted identity is authorized to make the invocation under the server configuration.

In this example, the following properties are set on the server side:

Property	Description
vbroker.security.disable=false	To enable the security service, set it to false.
vbroker.security.login=false	If set to true, at initialization-time this property tries to log on to all realms listed by property vbroker.security.login.realms.
vbroker.security.authentication.config=cpp_server.config	This specifies the path to the configuration file used for authentication.
vbroker.security.peerAuthenticationMode=none	This is to set the peer authentication mode. NONE—Authentication is not required. During handshake, no certificate request will be sent to the peer. Regardless of whether the peer has certificates, a connection will be accepted. There will be no transport identity for the peer.
vbroker.security.assertions.trust.1=Asserter@bank	This property is used to specify a list of trusted roles (specify with the format <role>@<authorization_domain>). <n> is uniquely identified for each trust assertion rule as a list of digits. For example, setting vbroker.security.assertions.trust.1=ServerAdmin@default means this process trusts any assertion made by the ServerAdmin role in the default authorization domain.
vbroker.security.authDomains=bank	Specifies a list of available authorization domains separated by comma.
vbroker.security.domain.bank.rolemap_path=cpp_bank.rolemap	Specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain <domain_name> specified in: vbroker.security.authDomains.

The following properties are set on the client side:

Property	Description
vbroker.security.server.transport=CLEAR_ONLY	It determines whether to use secure transport only or not. **Note:** To use secure transport only, the secureTransport property must also be set to true.
vbroker.security.login=true	If set to true, at initialization-time this property tries to login to all the realms listed by property vbroker.security.login.realms. On client side, this property is set to true,
vbroker.security.login.realms=myrealm	
vbroker.security.disable=false	To enable the security service, set it to false.

`vbroker.security.alwaysSecure=false`	This property determines whether to use secure transport only or not. **Note:** To use secure transport only, the `secureTransport` property must also be set to `true`.
`vbroker.security.peerAuthenticationMode=none`	This is to set the peer authentication mode. `NONE`—Authentication is not required. During handshake, no certificate request will be sent to the peer. Regardless of whether the peer has certificates, a connection will be accepted. There will be no transport identity for the peer.

Trusting Assertions

A server (end-tier) may choose to accept or not accept identity assertions.

In the case where it chooses to accept identity assertions, there are trust issues that present themselves. While the server may know that the peer is authentic, it must also confirm that the peer has the privilege to assert another caller or act on behalf of the caller. Since the caller itself is not authenticated by the end-tier, and the end-tier accepts the mid-tier's assertion, the end-tier needs to ensure that it trusts the mid-tier to have performed proper authentication of the original caller. It, in turn, trusts the mid-tier's trust in the authenticity of the caller.

There may be many peers to an end-tier system, some of whom are trusted as mid-tiers, and some that are just clients. Therefore, the privilege to speak for other callers must be granted only to certain peers.

Trust assertions and plug-ins

When a remote peer (server or process) makes identity assertions while acting on behalf of the callers, the end-tier server needs to trust the peer to make such assertions. The Security Provider Interface (SPI) allows you to plug in a Trust Services Provider to determine whether the assertion is allowed (trusted) for a given caller and a given set of privileges for the asserter. Specifically, you use the `TrustProvider` class to implement trust rules that determine whether the server will accept identity assertions from a given asserting subject. This allows you to plug in an assertion trust mechanism.

Assertion can happen in multi-hop scenario, or explicitly called through assertion API. The server can have rules to determine whether the peer is trusted to make the assertion or not. The default implementation uses property setting to configure trusted peers on the server side. During runtime, the peer must pass authentication and authorization in order to be trusted to make assertions.

Like the other pluggable services, you will need to define the authorization service with properties which are then passed as string maps. For example:

```
vbroker.security.trust.trustProvider=MyProvider
vbroker.security.trust.trustProvider.MyProvider.property1=xxx
vbroker.security.trust.trustProvider.MyProvider.property2=xxx
```

There can be only one trust provider specified for the whole security service.

For more information, refer to “vbsec::TrustProvider”

Backward trust

Backward trust is provided “out of the box”, and is the form of trust where the server has rules to decide who it trusts to perform assertions. With backward trust, the client has no say whether the mid-tier server has the privilege to act on its behalf.

Assertion example in the example folder is an example of backward trust.

Forward trust

Forward trust is similar to delegation in that the client explicitly provides certain mid-tier servers the privilege to act on its behalf. Forward Trust is currently not being supported in VisiSecure

Temporary privileges

At times, a server needs to access a privileged resource to perform a service for a client. However, the client itself may not have access to that privileged resource. Typically, in the context of an invocation, access to all resources are evaluated based on the original caller's identity. Therefore, it would not be possible to allow this scenario, as the original caller does not have access to such privileged resource. To support this scenario, the application may choose to assume an identity different from that of the caller, temporarily while performing that service. Usually, this identity is described as a *logical role*, as the application effectively requires to assume an identity that has access to all resources that require the user to be in that role.

The privileges class gives an abstraction of the privileges for a subject. It is the container of authorization privilege attributes, such as Distinguished Name (DN) attributes, and such. The AuthorizationService makes the decision on whether the subject has permission to access the certain resource based on the privileges object of the subject.

The privileges object is stored inside the subject as one of the PublicCredentials. At the same time, privileges hold one reference to the referring subject. Privileges also contain a DN attributes map, as well as a map of other.

4

Secure Transportation

In intranet scenarios, it may be safe to transfer information (including sensitive data, such as user authentication credentials) using IIOP over plain sockets. However, when the network environment is not trusted (such as the Internet, or even an intranet), you need to guarantee integrity (the message was not modified or tampered with during the transmission) and confidentiality (the message cannot be read by anybody even if they intercepted it during transmission) of messages being transmitted over the network. This is achieved by using secure sockets (SSL).

VisiSecure functions in two transport environments:

- using IIOP over plain sockets - (clear mode)
- using secure sockets (SSL) - (encrypted mode)

Enabling SSL

For Java only

VisiSecure uses Java Secure Sockets Extension (JSSE) to perform SSL communication. VisiSecure SPI Secure Socket Provider class provides access to the underline SSL implementation. Any appropriate implementation following Java Secure Socket Extension (JSSE) framework can be easily plugged in, independent of other provider mechanisms. The only necessary step is mapping the interfaces (or, in another word, callback methods) defined to the corresponding JSSE implementation. For more information on the SPI Secure Socket Provider class, see VisiSecure SPI for Java and “Security SPI for C++.”

For the “out-of-box” installation of VisiBroker, the JSSE implementation provided by Java SDK is used.

Setting the level of encryption

The SSL product uses a number of encryption mechanisms. These mechanisms are industry-standard combinations of authentication, privacy, and message integrity algorithms. This combination of characteristics is referred to as a *cipher suite*.

The client and server have a static list of supported cipher suites. This list is used during the handshake phase of the connection to determine which cipher suite will be used. The client sends a list of all cipher suites it knows to the server. The server then takes this information and determines which cipher suites both the server and client understand. By default, the server selects the strongest available cipher suite.

While this cipher suite order ensures strong security, you may want to adopt a different cipher suite order based on application-specific security requirements. When you want to change the order of the cipher suites, use the Quality of Protection (QoP) API function calls; you can retrieve a list of the currently available cipher suites, then set the list to a new order so weaker cipher suites are used before stronger cipher suites.

Note

You cannot add new cipher suites. You can modify only the order of the cipher suites that are available and remove cipher suites you do not want to use.

Supported cipher suites

A *cipher suite* is a set of valid encoding algorithms used to encrypt data. Cipher suites have different security levels and can serve different purposes. For example, some ciphers provide for authentication while others do not; some provide for encryption and others do not. Segments of the name of the cipher indicate what the cipher suite does or does not provide.

The following table shows the cipher name segments and what these segments mean.

Cipher name	Description
RC4 (through RC8)	Symmetric encryption used in the cipher
MD5	Data integrity mechanism. Data is sent clear, but a hash code is used at the receiving end to ensure data integrity.
SHA	Data integrity mechanism
WITH	Authentication with encryption
ANON	Uses DLT, an anonymous key exchange algorithm
NULL	No encryption
EXPORT	Public key size is limited. **Note:** The larger the size of a public/private key, the more secure that key is. This option is typically used for international (outside the United States) users.
EXPORT1024	The maximum key size is limited to 1024 bytes.

The list of supported ciphers for VisiSecure for Java, is determined by the JSSE package used. As for VisiSecure for C++, the list can be located at `csstring.h` file bundled with the installation.

Enabling Security

For a ORB to be secure, it must have the following property set:

```
vbroker.security.disable=false
```

Enabling SSL

To use SSL, your security in the ORB must be enabled. Once the security is enabled, ssl is enabled by default.

To disable the SSL:

To disable the SSL on the client side, set the property on the client side as given below.

```
Vbroker.security.secureTransport=false
```

To disable the SSL on the server side, you must set the property on the server side as given below.

```
vBroker.security.server.transport=CLEAR_ONLY
```

For more information on SSL, refer to “Enabling SSL”.

Setting the Log Level

Logging in VisiBroker employs one or more Logger objects. Applications can log messages to the Default Logger as well, to integrate their logging output with that of the ORB, or they can create one or more other Loggers, to log messages independently as said earlier.

All log messages to a single logger are bound to a common set of destinations. By using multiple loggers for logging, messages from different components could be made output to various independent end points.

ORB and all its C++ services use a special Logger instance (the 'Default Logger' with the name "default"), which is created automatically the first time the ORB logs a message. For more information, refer to the chapter on '*VisiBroker logging'* in the VisiBroker for C++ Developer's Guide.

`SimpleLogger` class is a mechanism to log information of various levels. Currently, it supports four different levels: `LEVEL_WARNING`, `LEVEL_NOTICE`, `LEVEL_INFO`, and `LEVEL_DEBUG`, with increasing detailed information. There is only one logger in the whole security service. For information on SimpleLogger class, refer to "vbsec::SimpleLogger".

VisiSecure - Java

Messages from VisiSecure java internal are logged under source name "secure".

For setting VisiSecure java logging messages to level info, set the following property to true.

```
vbroker.log.enable=true
```

```
vbroker.log.default.filter.secure.logLevel=info
```

The default value is "debug".

The VisiBroker Java logging mechanism applies to VisiSecure java as well.

VisiSecure C++

VisiBroker for C++ provides a logging mechanism, which allows applications to log messages and have them directed, via configurable logging forwarders called appenders, to appropriate destination or destinations. The ORB and all its services themselves use this mechanism for the output of any error, warning or informational messages.

The VisiBroker C++ logging mechanism applies to VisiSecure C++ as well.

For more information, refer to VisiBroker java logging in the Developer's Guide.

For setting VisiSecure csiv2 related logging messages, set the following property to true.

```
vbroker.log.enable=true
```

vbroker.log.default.filter.v_seccsiv2.logLevel=info

The default value is "debug".

Messages from VisiSecure C++ internal are logged under four separate source names as given below.

Types of Messages	Source Names
Authentication related messages	v_secauthn
Authorization related messages	v_secauthz
SSL related messages	v_secssl
CSIV2 related messages	v_seccsiv2

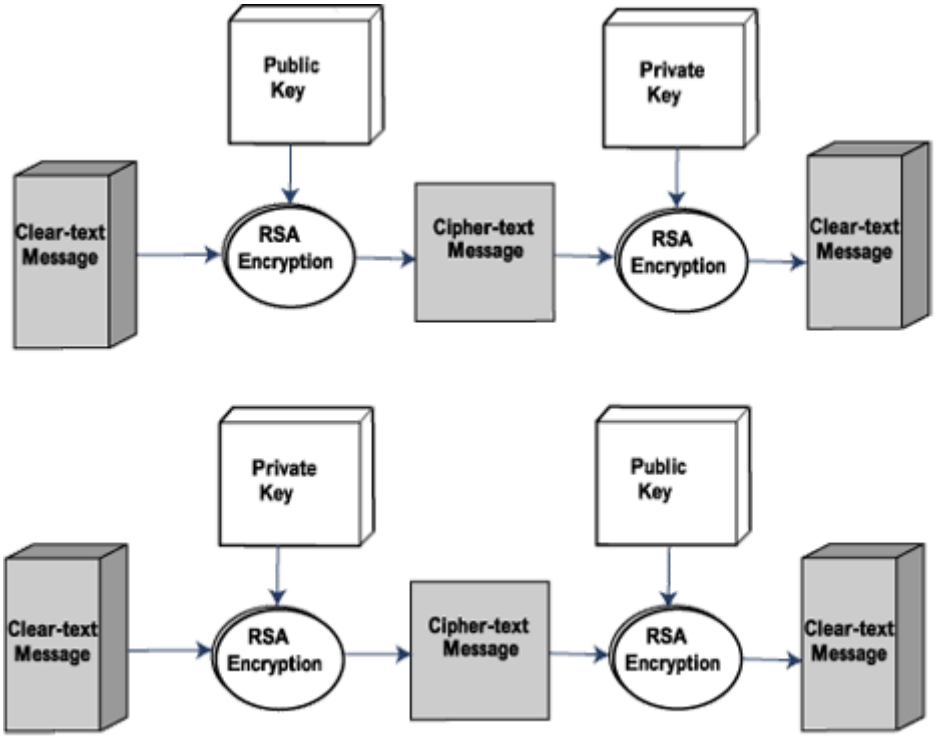
Encryption

Public-key encryption

In addition to username/password-based authentication, VisiSecure also supports *public-key encryption*. In public-key encryption, each user holds two keys: a *public key* and a *private key*. A user makes the public key widely available, but keeps the private key secret.

Data that has not been encrypted is often referred to as *clear-text*, while data that has been encrypted is called *cipher-text*. When a public key and a private key are used with the public-key encryption algorithm, they perform inverse functions of one another, as shown in the following diagram.

- In the first case, the public key is used to encrypt a clear-text message into a cipher-text message; the private key is used to decrypt the resulting cipher-text message.
- In the second case, the private key is used to encrypt a message (typically in the case of digital signatures—that is, “signed” messages), while the public key is used to decrypt it.



If someone wants to send you sensitive data, they acquire your public key and use it to encrypt that data. Once encrypted, the data can only be decrypted with the private key. Not even the sender of the data will be able to decrypt the data. The encryption can be *asymmetric* or *symmetric*.

Asymmetric encryption

Asymmetric encryptions has both a public and a private key. Both keys are linked together such that you can encrypt with the public key but can only decrypt with the private key, and vice-versa. This is the most secure form of encryption.

Symmetric encryption

Symmetric encryption uses only one key for both encryption and decryption. Although faster than asymmetric encryption, is requires an already secure channel to exchange the keys, and allows only a single communication.

Certificates and Certificate Authority

When you distribute your public key, the recipients of that key need some sort of assurance that you are indeed who you claim to be. The *ISO X.509 standard* defines a mechanism called a *certificate*, which contains a user's public key that has been digitally signed by a trusted entity called a *Certificate Authority* (CA). When a client application receives a certificate from a server, or vice-versa, the CA that issued the certificate can be used to verify that it did indeed issue the certificate. The CA acts like a notary and a certificate is like a notarized document.

You obtain a certificate by constructing a certificate request and sending it to a CA.

Digital signatures

Digital signatures are similar to handwritten signatures in terms of their purpose; they identify a unique author. Digital signatures can be created through a variety of methods. Currently, one of the more popular methods involves an encrypted hash of data.

1. The sender produces a one-way hash of the data to be sent.
2. The sender digitally signs the data by encrypting the hash with a private key.
3. The sender sends the encrypted hash and the original data to the recipient.
4. The recipient decrypts the encrypted hash using the sender's public key.
5. The recipient produces a one-way hash of the data using the same hashing algorithm as the sender.
6. If the original hash and the derived hash are identical, the digital signature is valid, implying that the document is unchanged and the signature was created by the owner of the public key.

Generating a private key and certificate request

To obtain a certificate to use in your application, you need to first generate a private key and certificate request. To automate this process, for Java applications you can use the Java keytool, or for C++ applications you can use open source tools like `OpenSSL` utility.

After you generate the files, you should submit the certificate request to a CA. The procedure for submitting your certificate request to a CA is determined by the certificate authority which you are using.

If you are using a CA that is internal to your organization, contact your system administrator for instructions.

If you are using a commercial CA, you should contact them for instructions on submitting your certificate request.

The certificate request you send to the CA will contain your public key and your distinguished name.

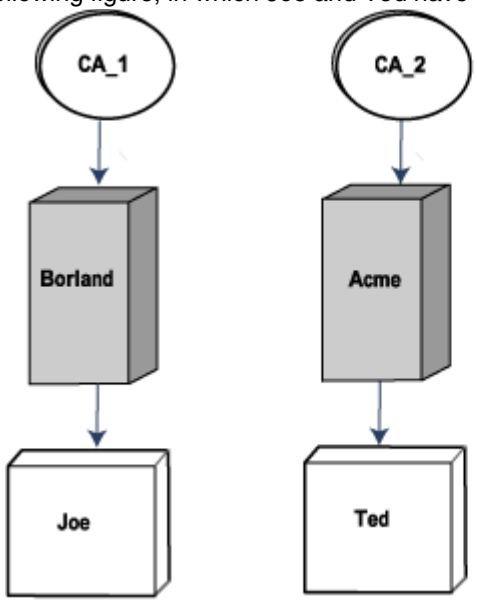
Distinguished names

A *distinguished name* represents the name of a user or the CA that issued the user's certificate. When you submit a certificate request, it includes a distinguished name for the user that is made up of the components listed in the following table.

Tag	Description	Required Component
Common-Name	The name to be associated with the user.	Yes
Organization	The name of the user's company or organization.	Yes
Country	The two character country code that identifies the user's location.	Yes
Email	The person to contact for more information about this user.	No
Phone	The user's phone number.	No
Organizational Unit	The user's department name.	No
Locality	The city in which the user resides.	No

Certificate chains

The ISO X.509 standard provides a mechanism for peers who wish to communicate, but whose certificates were issued by different certificate authorities. Consider the following figure, in which Joe and Ted have certificates issued by different CAs.



For Joe to verify the validity of Ted's certificate, he must inspect each CA in the chain until a trusted CA is found. If a trusted CA is not found, it is the responsibility of the server to choose whether to accept or reject the connection. In the case shown in the preceding figure, Joe would follow these steps:

1. Joe obtains Ted's certificate and determines the issuing CA, `Acme`.
2. Since the `Acme` CA is not in Joe's certificate chain, Joe obtains the issuer of the certificate for `CA_2`.
3. Because `CA_2` is not a trusted CA, the server decides whether to accept or reject the connection.

Note

The manner in which you obtain certificate information from a CA is defined by that CA.

Using IIOP/HTTPS

VisiBroker has a feature that allows tunneling of IIOP inside the HTTP protocol. This is an extended feature in VisiBroker called HIOP. With VisiSecure enabled, the secure version of HIOP is available. This allows tunneling of IIOP inside HTTPS.

You can make use of HTTPS, featured in most browsers. The following guidelines should be followed:

- The VisiBroker proxy server GateKeeper must be running with SSL enabled on the exterior.
- An applet that only uses IIOP/HTTPS requires no pre installation of software (either classes or native libraries) on the client as long as the browser or applet viewer is HTTPS enabled.
- An applet using IIOP/HTTPS cannot use the `X509Certificate[]` class to set or examine identities. All certificate and private key administration is handled by the browser. Furthermore, when the `ORBalwaysTunnel` parameter in the applet tag is set to `true`, the ORB cannot resolve `SSLCurrent` objects.
- To enable an applet to use only IIOP/HTTPS, set `ORBalwaysTunnel` to `true` in the HTML page. If `ORBalwaysTunnel` is set to `false` (or unspecified) the ORB first tries to use IIOP/SSL, which requires the SSL classes and native SSL library to be installed locally.
- In general, IIOP/HTTPS is not available to Java applications because HTTPS is not supported by the JDK. However, there are no restrictions in VisiBroker for Java that prevent the addition of HTTPS support to the JDK and the use of IIOP/HTTPS in Java applications.

Netscape Communicator/Navigator

You can freely use Netscape Communicator with IIOP/HTTPS, however, some versions of Navigator require the installation of the CA certificate before allowing an IIOP/HTTPS connection. Follow these guidelines to use IIOP/HTTPS with Netscape Navigator:

- Make sure your server certificates are issued by a CA already trusted by Navigator.
- Install the root certificate into Navigator as a trusted certificate. Opening a certificate file (for example, cacert.crt in bank_https) gives you the opportunity to install the certificate.
- Use the GateKeeper to download the root certificate to the browser. The bank_https example shows how to do this.
- Commercial CAs usually provide a link that allows you to install their root certificate.
- GateKeeper, by default, does not ask for the client identity. You can enable this function by setting ssl_request_client_certificate to true in the GateKeeper configuration file.

Microsoft Internet Explorer

To use IIOP/HTTPS with Microsoft Internet Explorer, you must make sure that the HTTPS connection requires no user interaction. For example, if the browser visits a HTTPS site with an untrusted root certificate, the browser will ask for permission before establishing an HTTPS connection. The Microsoft JVM, due to a known bug, fails on this connection.

Here are several examples that illustrate this condition and ways in which you can work:

- Internet Explorer ships with a list of trusted Network Server Certificates Authority. If your server certificate is not issued by one of the trusted CAs, (the certificates shipped with bank_https, for example) IE asks for permission before establishing an HTTPS connection. The IIOP/HTTPS operation fails because the Microsoft JVM does not seem to support an HTTPS connection that requires user interaction. There are a number of ways to handle this situation:
  - Make sure your server certificates are issued by a CA already trusted by Internet Explorer.
  - Install the root certificate into IE as a trusted Network Server certificate. Opening a certificate file (for example, cacert.crt in bank_https) gives you the opportunity to install the certificate.
  - Use the GateKeeper to download the root certificate to the browser. The bank_https example shows how to do this.
  - Commercial CAs usually provide a link that allows you to install their root certificate.
- GateKeeper, by default, does not ask for the client identity. Although, you can enable this function by setting ssl_request_client_certificate=true in the GateKeeper configuration file, you cannot use IIOP/HTTPS because the browser asks for permission before responding with the user's credentials.

Internet Explorer optionally requires the Common Name field within the server certificate to be the same as the host name of the server.

- From the `View|Internet` Options menu, select the `Advanced` tab and scroll to the `Security` section.
- Make sure the box next to `Warn` about is invalid.
- Make sure the site certificates are not checked. If checked, it would allow using a server certificate that does not contain the host name of the server.

5

Quality of Protection

Quality of Protection (QoP) is an OMG policy. It enforces some requirement on the run-time and it is installed on a POA. Server QOP must be installed on a POA. Client QOP can be installed on an object.

The Portable Object Adapter (POA) is the piece of the ORB that manages server-side resources for scalability. By deactivating objects' servants when they have no work to do, and activating them again when they're needed, we can stretch the same amount of hardware to service many more clients.

Although it is architecturally a separate piece of the ORB, and ORBs can have more than one type of object adapter, the POA is not a piece of software that you can buy separately from the ORB. In fact, the interfaces that connect the ORB and the POA are proprietary. The POA is the second object adapter that OMG has specified. First was the Basic Object Adapter or BOA, now deprecated.

The programmer fixes resource allocation/de-allocation patterns by setting POA Policies. These determine whether object references created by the POA are transient or persistent; whether activation is per-method-call or longer or shorter; and how multiple CORBA objects (either of a single type, or multiple types) map to servants. There are seven POA policies; when you multiply out the different settings, you come up with nearly 200 possible

Setting properties and QoP

There are several properties that can be used to ensure connection Quality of protection. The VisiBroker ORB security properties for C++ can be used to fine tune connection quality.

For example, you can set the cipherList property for SSL connections to set cryptography strength.

```
vbroker.security.cipherList
```

This property is set to a list of comma-separated ciphers to be enabled by default on startup. If not set, a default list of ciphersuites will be enabled. There should be valid SSL Ciphers.

QOP properties can be set using `ServerQoPConfig` and the `ClientQoPConfig` for servers and clients, repectively. For more information, see “Configuring Quality of Protection(QoP) for both server and the client.”.

These APIs allow you to set target trust (whether or not targets must authenticate), the transport policy (whether or not to use SSL or another secure transport mechanism specified separately). For servers, an AccessPolicyManager that can access the

RoleDB is set to access policies for POA objects. For more information on AccessPolicyManager, see “class csiv2::AccessPolicyManager”

Configuring Quality of Protection(QoP) for both server and the client.

Configuring QoP for Server

The complete code of ServerQoPConfigValueFactory is as follows,

```
package com.borland.security.csiv2;
import com.borland.security.csiv2.ServerQoPConfigValueFactory;
import com.borland.security.csiv2.ServerQoPConfig;
import com.borland.security.csiv2.AccessPolicyManager
public class ServerQoPConfigDefaultFactory
implements ServerQoPConfigValueFactory {
public ServerQoPConfig createConfig ( boolean disable,
                                      short transport,
                                      short idType,
                                      boolean
                                      enableIdAssertion,
                                      java.lang.String[]
realms, AccessPolicyManager access_manager )
{
return new ServerQoPConfigImpl(disable, transport, idType, enableIdAssertion, realms,
access_manager);
  }
}
disable = security is disabled/enabled for this POA, When security is disabled, the rest
of the settings become irelevant.
```

The transport methods has three possible values of CLEAR_ONLY or SECURE_ONLY or ALL of `com.borland.security.csiv2.ServerQoPPolicyOperations`.

where `ServerQoPPolicyOperations` is the class.

CLEAR_ONLY: Uses only clear listener to accept request

SECURE_ONLY: Uses only SSL listener to accept request

ALL: Uses both

The `IdType` has possible values of `com.borland.security.csiv2.ServerQoPPolicy`.

`NO_ID`	means expecting no identity
`UP`	means expecting Username Password identity
`PK`	means expecting transport identity
`UP_AND_PK`	means expecting both must present and valid
`UP_OR_PK`	means expecting either one must present and valid

`enableIdAssertion = true/false`, when this is set to false, this server can not accept caller identity propagated through CSIV2 Authorization token.

`realms[]` = is an array of strings, specifying the names of all realms that this POA can accept identity of. The default value is ‘null’ meaning there are no configured realms in this ORB.

`access_manager` = For authorization purposes, this is the AccessPolicyManager responsible for this POA; The default value is ‘null’ meaning there is no authorization.

For configuring Quality of Protection(QoP) for server, follow the steps as given below:

For server, QoP is set as follows:

To enable access controls, set `disable = false`

`For method to be secure, set transport = secure only`
For server to require clients credentials for authentication, set `trust in client = true`.

For more information on transport methods and other QoP related parameters, see "class vbsec::ServerConfigImpl"

1 To create server QoP configuration object

```
ServerQoPConfig config = new
ServerQoPConfigDefaultFactory().create(false,ServerQoPPolicy.SECURE_ONLY,true, null);
```

2 To activate the server with QoP

```
Any any = orb.create_any();
ServerQoPConfigHelper.insert(any, config);
Policy qop = orb.create_policy(SERVER_QOP_CONFIG_TYPE.value, any);
```

Configuring QoP for client

The initial step of creating a QoP is to create a QoPConfig and specify the security requirements that must be enforced through the config.

To create a `ClientQoPConfig`, you can use its default factory as follows:

```
...
   ...
      com.borland.security.csiv2.ClientQoPConfig myconfig =
      new com.borland.security.csiv2.ClientQoPConfigDefaultFactory().create
/* transport    = */ com.borland.security.csiv2.ClientQoPPolicyOperations.CLEAR_ONLY,
     /* Other possible values for above are SECURE_ONLY and USE_ANY*/
      /* trustInTarget= */ true
      );
```

Transport methods	trustInTarget=true	trustInTarget=false
CLEAR_ONLY	Invalid conf, will throw PolicyError	For outgoing, use clear IIOP transport
SECURE_ONLY	For outgoing, use SSL and make sure that server cert is trusted	For outgoing, use SSL and server cert can be non trusted
USE_ANY	Invalid conf, will throw PolicyError	For outgoing, try SSL then fallback to clear when fail, server cert can be non trusted

The complete code of `ClientQoPConfigDefaultFactory` is as follows,

```
package com.borland.security.csiv2;

public class ClientQoPConfigDefaultFactory
      implements com.borland.security.csiv2.ClientQoPConfigValueFactory {
   public com.borland.security.csiv2.ClientQoPConfig create ( short trans,
                                                              boolean trustInTarget ) {
      return new ClientQoPConfigImpl( trans, trustInTarget );
    }
   }
```

1. Initialize the ORB/

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
   byte[] managerId = "BankManager".getBytes();
```

Note: For client, QoP is set as follows:

```
For method to be secure, set transport = secure only
```

Set `trust in target = false`. (With this, server need not provide authentication for client).

For more information on transport methods and other QoP parameters, see "class vbsec::ClientConfigImpl"

```
ClientQoPConfig cc = new ClientQoPConfigDefaultFactory().create(
ClientQoPPolicy.SECURE_ONLY, false);
org.omg.CORBA.Object managerObject =
Bank.AccountManagerHelper.bind( orb, "/bank_agent_poa", managerId );
```

2. Insert client QoP into any object

```
Any any = orb.create_any();
ClientQoPConfigHelper.insert( any, cc);
```

3. Narrow the reference to Bank Manager using the policy

```
      Bank.AccountManager manager = Bank.AccountManagerHelper.narrow(
         managerObject._set_policy_override(
            new Policy[] { orb.create_policy( CLIENT_QOP_CONFIG_TYPE.value, any)
},
   SetOverrideType.SET_OVERRIDE));
```

4. Create or open an account for the user. This returns a generic Account object reference.

```
      org.omg.CORBA.Object accountObject = manager.open( name);
```

5. Narrow the reference to Bank Account

```
      Bank.Account account = Bank.AccountHelper.narrow(
         accountObject._set_policy_override(
            new Policy[] { orb.create_policy( CLIENT_QOP_CONFIG_TYPE.value, any) },
      SetOverrideType.SET_OVERRIDE
)
   );
```

6. Get the balance from the account and print it on the console

```
      System.out.println("The balance in " + name + "'s account is $" +
account.balance() );
   } catch( Throwable e) {
      synchronized( System.err) {
         e.printStackTrace();
      }
         }
            }
public static void main(String[] args) {
    if ( args.length != 0) {
      System.err.println( "Usage : vbj -DORBpropStorage=client.properties Client");
      System.exit(1);
    }
    Client cln = new Client();
    cln.test(args);
  }
}
```

Configuring Quality of Protection (QoP) parameters

When clients and servers communicate, they both need to agree on certain parameters for the Quality of Protection (QoP) that will be provided. The resource host (the server) will:

- publish all the QoP parameters that it can support, and
- impose a set of required QoP parameters on the clients.

Note

By definition, a required QoP is also a supported QoP.

For example, a server may support and require secure transport (SSL) while it may support authentication but not require it. This is useful, for example, in the case where some resources are not sensitive and anonymous access is acceptable. For more information about QoP and QoS parameters:

For C++, See “QoP API”.

For Java, See `com.borland.security.csiv2` in ..//VisiBroker/doc/sec-api-doc/com/borland/security/csiv2/package-frame.html.

6

Creating Custom Plugins

Creating Custom Plugins for c++

There are various components of VisiSecure that allow for custom plug-ins. They are:

- LoginModules
- CallBack Handlers
- Authorization service provider via the SPI
- Assertion Trust via the SPI

In order for VisiSecure for C++ to find user implementations, all plugins must use the `REGISTER_CLASS` macro provided by VisiSecure to register their classes to the security service. When specifying the registered class, the name of the class must be specified in full together with the name space. Name spaces must be specified in a normalized form, with either a "." or "::" separated string starting from the outermost name space. For example:

```
MyNameSpace {
  class MyLoginModule {
    ......
  }
}
```

would be specified as either `MyNameSpace.MyLoginModule` or `MyNameSpace::MyLoginModule`.

LoginModules

LoginModule describes the interface implemented by authentication technology providers. LoginModules are plugged under applications to provide a particular type of authentication.

While applications write to the LoginContext API, authentication technology providers implement the LoginModule interface. A Configuration specifies the LoginModule(s) to be used with a particular login application. Therefore different LoginModules can be plugged in under the application without requiring any modifications to the application itself.

You can implement your own LoginModules by extending `vbsec::LoginModule`.

`LoginModule` serves as the parent of all login modules. User plugin login modules must extend this class. Login modules are configured in the authentication configuration file and called during the login process. Login modules are responsible of authenticating the given subject and associating relevant Principals and Credentials with the subject. It is also responsible for removing and disposing of such security information during logout.

To use the LoginModule, you need to set it in the authentication configuration file, just like any other LoginModule. During runtime, the new customized module will need to be loaded by the secured application.

The syntax of the authentication configuration is as follows:

```
<realm-name> {
    <class-name-of-customized-LoginModule> <required|optional>;
}
```

Note: There is implicit replacement of the character "." to "::" by VisiSecure. Hence, `com.borland.security.provider.authn.HostLoginModule` is equivalent to `com::borland::security::provider::authn::HostLoginModule`.

For more information, see "vbsec::LoginModule".

The first thing you need to do is to determine whether or not your LoginModule will require some form of user interaction (retrieving a user name and password, for example). If so, you will need to become familiar with the CallbackHandler interfaces readily available. (Alternatively, you can create your own Callback implementations.)

The LoginModule will invoke the CallbackHandler specified by the application itself and passed to the LoginModule's initialize method. The LoginModule passes the CallbackHandler which is an array of appropriate Callbacks.

If the LoginModule implementations have no end-user interactions, the LoginModules would not need to access the callback package.

You must also determine what configuration options you want to make available to the user, who specifies configuration information in whatever form the current Configuration implementation expects (for example, in files). For each option, decide the option name and possible values.

For example, if a LoginModule may be configured to consult a particular authentication server host, decide on the option's key name ("auth_server", for example), as well as the possible server hostnames valid for that option.

To implement the login module, you first have to decide on the proper package and class name for your `LoginModule`.

The LoginModule interface specifies five abstract methods that require implementations: initialize, login, commit, abort, logout

For more information on implementing login modules, see *Login Module Developer's Guide* in *Sun JDK - JAAS Documentation*.

In addition to these methods, a LoginModule implementation must provide a public constructor with no arguments. This allows for its proper instantiation by a LoginContext.

Note: If no such constructor is provided in your LoginModule implementation, a default no-argument constructor is automatically inherited from the Object class.

The LoginContext is responsible for reading the configuration and instantiating the appropriate LoginModules. Each LoginModule is initialized with a subject, a CallbackHandler, shared LoginModule state, and LoginModule-specific options. The subject represents the subject currently being authenticated and is updated with relevant credentials if authentication succeeds.

The LoginModule-specific options represent the options configured for this LoginModule by an administrator or user in the login configuration. The options are defined by the LoginModule itself and control the behavior within it.

The calling application sees the authentication process as a single operation. However, the authentication process within the LoginModule proceeds in two distinct phases.

In the first phase, the LoginModule's login method gets invoked by the LoginContext's login method. The login method for the LoginModule then performs the actual authentication (prompt for and verify a password for example) and saves its authentication status as private state information. Once finished, the LoginModule's login method either returns true (if it succeeded) or false (if it should be ignored), or

throws a LoginException to specify a failure. In the failure case, the LoginModule must not retry the authentication or introduce delays. The responsibility of such tasks belongs to the application. If the application attempts to retry the authentication, the LoginModule's login method will be called again.

In the second phase, if the LoginContext's overall authentication succeeded (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules succeeded), then the commit method for the LoginModule gets invoked. The commit method for a LoginModule checks its privately saved state to see if its own authentication succeeded. If the overall LoginContext authentication succeeded and the LoginModule's own authentication succeeded, then the commit method associates the relevant principals (authenticated identities) and credentials (authentication data such as cryptographic keys) with the subject located within the LoginModule.

If the LoginContext's overall authentication failed (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules did not succeed), then the abort method for each LoginModule gets invoked. In this case, the LoginModule removes/destroys any authentication state originally saved.

Logging out a subject involves only one phase. The LoginContext invokes the LoginModule's logout method. The logout method for the LoginModule then performs the logout procedures, such as removing principals or credentials from the subject or logging session information.

A LoginModule implementation must have a constructor with no arguments. This allows classes which load the LoginModule to instantiate it.

CallbackHandlers

`CallbackHandler` is the mechanism that produces any necessary user callbacks for authentication credentials and other information. Callbacks are an array of callback objects which contain the information requested by an underlying security service that has the ability to interact with a calling application to retrieve specific authentication data such as usernames and passwords, or to display certain information, such as errors and warning messages.

The CallbackHandler may be used to prompt for usernames and passwords, for example. Note that the CallbackHandler may be null. LoginModules which absolutely require a CallbackHandler to authenticate the subject may throw a LoginException. LoginModules optionally use the shared state to share information or data among themselves.

Underlying security services make requests for different types of information by passing individual callbacks to the CallbackHandler. The CallbackHandler implementation decides how to retrieve and display information depending on the callbacks passed to it.

For example, if the underlying service needs a username and password to authenticate a user, it uses a NameCallback and PasswordCallback. The CallbackHandler can then choose to prompt for a username and password serially, or to prompt for both in a single window.

There are seven types of callbacks provided. There is a default handler that handles all callbacks in interactive text mode.

You can implement your own callback by extending `vbsec::CallbackHandler`. To use the callback, you need to set the property `vbroker.security.authentication.callbackHandler=<custom-handler-class-name>` in the security property file, just like any other callback handler.

This property specifies the callback handler that is used by login modules for interacting with users for credentials. You can specify one of the following or your own custom callback handler. Refer to C++ Security API for more details.

`com.borland.security.provider.authn.CmdLineCallbackHandler`

`com.borland.security.provider.authn.HostCallbackHandler`

`CmdLineCallbackHandler` has password echo on, while `HostCallbackHandler` has password echo off.

During runtime, the new customized module will need to be loaded by the secured application.

For more information on implementing call back handlers, see “vbsec::CallbackHandler”.

Implementations of the calback interface are passed to a CallbackHandler, allowing underlying security services that has the ability to interact with a calling application to retrieve specific authentication data such as usernames and passwords, or to display certain information, such as error and warning messages.

Callback implementations do not retrieve or display the information requested by underlying security services. Callback implementations simply provide the means to pass such requests to applications, and for applications, if appropriate, to return requested information back to the underlying security services.

Authorization Service Provider

Authorization is the process of making access control decisions on behalf of certain resources based on security attributes or privileges. VisiSecure uses the notion of Permission in authorization. The class `RolePermission` is defined to represent a “role” as a permission. Authorization Services Providers in turn provide the implementation on the homogeneous collection of role permissions that associate privileges with particular resources.
The implementer of the Authorization Service provides the collection of permission objects granted access to certain resources. Whenever an access decision is going to be made, the AuthorizationServicesProvider is consulted. The Authorization Service is closely associated with the Authorization domain concept. An Authorization Service is installed per each Authorization domain implementation, and functions only for that particular Authorization domain.

The `AuthorizationServicesProvider` is initialized during the construction of its corresponding Authorization domain.

Use the following property to set the implementing class for the AuthorizationServicesProvider:

```
vbroker.security.domain.<domain-name>.provider
```

During the runtime, this property is loaded by way of Java reflection.

Another import functionality of the Authorization Service is to return the run-as alias for a particular role given. The security service is configured with a set of identities, identified by aliases. When resources request to “run-as” a given role the AuthorizationServices again is consulted to return the alias that must be used to “run-as” in the context of the rules specified for this authorization domain.

Authorization service providers are tightly connected with Authorization Domains. Each domain has exactly one authorization service provider implementation. During the initialization of the ORB, the authorization domains defined by `vbroker.security.authDomains` is constructed, while the Authorization Service Provider implementation is instantiated during the construction of domain itself.

To plugin authorization service, you need to set the following properties:

```
vbroker.security.auth.domains=MyDomain
vbroker.security.domain.MyDomain.provider=MyProvider
vbroker.security.domain.MyDomain.property1=xxx
vbroker.security.domain.MyDomain.property2=xxx

vbroker.security.identity.attributeCodecs=MyCodec
vbroker.security.adapter.MyCodec.property1=xxx
vbroker.security.adapter.MyCodec.property2=xxx
```

The properties specified will be passed to the user plugin following the same mechanism as above.

Example:

You can write a custom authorization module using both user names and groups. Use a HostLoginModule, since that is the only supported login module for C++ Visibroker security applications. HostLoginModule must show the configurations required and the code required for corba components to use the authorization framework. The Client needs to have these authorizations in order to access these components in the server.

The roles must be hardcoded into the authorization provider code. The groups for a user can also be obtained from a different source programmatically and the subject can

be populated with groups as privileges added to the public credentials of the subject in question, at runtime, for use by VisiSecure authorization mechanism.

You can match the user/group with roles obtained from an external source(for example, a legacy system) other than the Borland rolemap mechanism.

USE_STD_NS is a define setup by VisiBroker to use the std namespace

USE_STD_NS

```
typedef pair<std::string, std::string> String_String_Pair;
typedef pair< std::string, std::set<std::string> > String_Set_Pair;
const std::string USE_CALLER_IDENTITY = (const char*)"use-caller-identity";
const std::string RUNAS = (const char*)"runas.";
void CustomProviderImpl::initialize (const std::string& name, vbsec::InitOptions& opts)
{
```

To store name of the module:

```
    _name = name;
    cout << "Custom authorization service Provider : " << name << endl;
    custProvider = this;
```

To print out the options given to the custom autorization service provider

```
    std::map<std::string, std::string> t_options;
    std::map<std::string, std::string>::iterator itr;
    std::basic_string <char>::size_type index1;
    static const std::basic_string <char>::size_type npos = -1;
    std::basic_string<char> t_key, ts;

    t_options = *(opts.options);
    for ( itr = t_options.begin(); itr!=t_options.end(); itr++ ) {
    cout << "Options key :" << itr->first << ", value : " << itr->second << endl;
    t_key = (itr->first).substr(0,RUNAS.size()-1);
    if ( t_key == RUNAS ) {
    cout << "runas property found :" << itr->first << endl;;
          ts = itr->first;
          index1 = ts.find_last_of(".", ts.size()-1);
          if ( index1 != npos )
             ts = ts.substr( index1+1, ts.size()-1 );
          else
             ts = "";
          cout << "runas role : " << ts << endl;
          if ( itr->second == USE_CALLER_IDENTITY )
             _callerRunAsRoles.insert(ts);
          else
             _runAsMap.insert(String_String_Pair(ts, itr->second) );
       }
     }
```

To store the logger reference:

```
_logger = opts.logger;
```

To store logLevel:

```
_logLevel = opts.logLevel;
```

You can also use an in-memory role table. To create the in-memory DB that holds the users and groups, do the following:

```
     createInMemoryDB();
```

```
return;
}
vbsec::PermissionCollection* CustomProviderImpl::getPermissions(const vbsec::Resource* res,
const vbsec::Privileges* prv)
{
   CustomProviderImpl::CustomPermissionCollectionImpl* perm_coll = new
CustomProviderImpl::CustomPermissionCollectionImpl();
   perm_coll->init( (vbsec::Privileges*)prv );
   return ( (vbsec::PermissionCollection*) perm_coll );
}
std::string CustomProviderImpl::getRunAsAlias(const std::string& s)
{
   std::string s1;
        std::map<std::string, std::string>::iterator it;
      std::map< std::string, std::set<std::string> >::iterator it2;
        std::set<std::string>::iterator it3;

        it = _runAsMap.find(s);
        if ( it == _runAsMap.end() ) {
           it2 = _inMemoryDB.find(s);
           if ( it2 == _inMemoryDB.end() )
            throw CORBA::NO_PERMISSION((CORBA::ULong)0x56422501,
CORBA::CompletionStatus::COMPLETED_NO, (const char*)"The RunAs Role specified does not
exist");
           it3 = _callerRunAsRoles.find(s);
           if ( it3 != _callerRunAsRoles.end() )
              s1 = USE_CALLER_IDENTITY;
           else
              s1 = (const char*)"";
}
        else
          s1 = it->second;
      return s1;
}
void CustomProviderImpl::createInMemoryDB()
{
```

For example,

If the authorization requirement for the BankManager object is that the clients should be member of the "Manager" role and for the Account object, it is "Customer" or "Teller" role.

- To create role entry "Manager" along with its user(s) and/or group(s) as a set:

```
 _role1_ug.insert("jjagadeesan"); // user "jjagadeesan"
_role1_ug.insert("FI.PSO");  // group "FI.PSO"
_inMemoryDB.insert(String_Set_Pair("Manager", _role1_ug));
```

- To create role entry "Customer" along with its user(s) and/or group(s) as a set:

```
   _role2_ug.insert("admin"); // user "admin"
   _inMemoryDB.insert(String_Set_Pair("Customer", _role2_ug));
```

- To create role entry "Teller" along with its user(s) and/or group(s) as a set:

```
   _role3_ug.insert("admin"); //user "admin"
   _role3_ug.insert("user");  // group "user"
```

```
    _inMemoryDB.insert(String_Set_Pair("Teller", _role3_ug));
}
   std::set<std::string>* CustomProviderImpl::getRoleEntries( std::string& role )
{
       std::set<std::string> * roleEntries;
      std::map< std::string, std::set<std::string> >::iterator it;

       it = _inMemoryDB.find( role );
      if ( it == _inMemoryDB.end() ) {
          roleEntries = new std::set<std::string>();
          roleEntries->clear();
      }
      else {
          roleEntries = new std::set<std::string>(it->second);
      }
      return roleEntries;
}
```

Implementation of the functions of the CustomePermissionCollectionPmpl class

```
void CustomProviderImpl::CustomPermissionCollectionImpl::init( vbsec::Privileges *prv )
{
   _privileges = prv;
   _provider = CustomProviderImpl::custProvider;
}
bool CustomProviderImpl::CustomPermissionCollectionImpl::implies (const ::vbsec::Permission&
p) const
{
   bool matchedRole = false;
   std::string userName;
   std::string groupName;
   string s = p.getName();
   // if(_logLevel >= 5 )
   //   _logger.notice( null, "Permission: " + s );
   cout << "In CustomAuthorizationProvider::implies: Permission role: " << s << endl;
   vbsec::Privileges *privileges = _privileges;
   vbsec::Subject& subject = privileges->getSubject();
   std::set<vbsec::Principal *> principals = subject.getPrincipals();
   std::multimap<std::string, std::string> groupMap = privileges->getAttributes();
   std::multimap<std::string, std::string>::iterator it_groups;
   std::set<std::string> groups;
   it_groups = groupMap.find("group");
   while ( it_groups != groupMap.end() ) {
      if ( it_groups->first == "group" ) {
         groups.insert( it_groups->second );
         break;
      }
      ++it_groups;
   }
   std::set<std::string> * roleEntities = _provider->getRoleEntries(s );
```

- To check the given role for existence in the internal table:

```
   if ( !roleEntities )
   {
```

```
      cout << "In CustomAuthorizationProvider::implies: Role: " << s << " not found in roles
table" << endl;
      return false;
   }
   if( roleEntities->empty() )
   {
      cout << "In CustomAuthorizationProvider::implies: Role: " << s << " not found in roles
table" << endl;
      delete roleEntities;
      return false;
   }
```

- To check if one of the principals matched for role

```
   if ( principals.empty() )
   {
      delete roleEntities;
      return false;
   }
   std::set<vbsec::Principal *>::iterator i;
   std::set<std::string>::iterator i_set_str, i_set_str2;
   for ( i = principals.begin(); i != principals.end(); i++ ) {
      vbsec::UserPrincipal *up = dynamic_cast<vbsec::UserPrincipal*>( *i );
      userName = up->getUserName();
      cout << "In CustomAuthorizationProvider::implies: Checking for username match: " <<
userName << endl;
      i_set_str = roleEntities->find( userName );
      if ( i_set_str != roleEntities->end() ) {
        cout << "In CustomAuthorizationProvider::implies: Found role entry for username:" <<
userName << endl;
         delete roleEntities;
         return true;
      }
   }
```

- To check now if at least a user group can be found for the role

```
   if ( groups.empty() )
   {
      delete roleEntities;
      return false;
   }
   for ( i_set_str = groups.begin(); i_set_str != groups.end(); i_set_str++ ) {
      groupName = (*i_set_str );
      cout << "In CustomAuthorizationProvider::implies: Checking for groupname match: " <<
groupName << endl;
      i_set_str2 = roleEntities->find( groupName );
      if ( i_set_str2 != roleEntities->end() ) {
         cout << "In CustomAuthorizationProvider::implies: Found role entry for groupname:"
<< groupName << endl;
         delete roleEntities;
         return true;
      }
   }
   delete roleEntities;
   return false; // all match failed
```

```
}
****************************
#ifndef _CUSTOMPROVIDER_H_
#define _CUSTOMPROVIDER_H_
#include "vbauthz.h"
#include <map>
#include <set>
#include <hash_map>
#include <string>
#include <iostream>
// typedef pair<std::string, std::string> String_String_Pair;
// typedef pair<std::string, std::set> String_Set_Pair;
// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS
class CustomProviderImpl : public vbsec::AuthorizationServiceProvider
{
    class CustomPermissionCollectionImpl : public vbsec::PermissionCollection
   {
       public:
      CustomPermissionCollectionImpl() {}
       void init( vbsec::Privileges* prv );
      virtual bool implies (const vbsec::Permission& p) const;
      virtual ~CustomPermissionCollectionImpl () {}
   private:
   vbsec::Privileges* _privileges;
         CustomProviderImpl* _provider;
   };
   public:
CustomProviderImpl() : _logLevel((int)0), _name(""), _logger((vbsec::SimpleLogger*)NULL)
    {}
      virtual std::string getName() const
    {
       return _name;
    }
      virtual void initialize (const std::string& name, vbsec::InitOptions& opts);
   vbsec::PermissionCollection* getPermissions(const vbsec::Resource* res, const
vbsec::Privileges* prv);
   std::string getRunAsAlias(const std::string& s);
   void createInMemoryDB();
   std::set<std::string>* getRoleEntries( std::string& role );
   static CustomProviderImpl * custProvider;
private:
    CORBA::ULong _logLevel;
    ::vbsec::SimpleLogger* _logger;
   std::map<std::string, std::string> _runAsMap;
    std::set<std::string> _callerRunAsRoles;
    std::string _name;
   std::map<std::string, std::set<std::string> > _inMemoryDB; // contains the known roles
    std::set<std::string> _role1_ug, _role2_ug, _role3_ug; // contains the users and/or
groups for the roles in _inMemoryDB
};
```

```
REGISTER_CLASS(CustomProviderImpl)
CustomProviderImpl * CustomProviderImpl::custProvider;
#endif
```

You can secure your application using VisiBroker properties and JAAS configuration file to secure your application. The example client and server uses username/password authentication of the client on the server and also for server's self authentication.

Look at the different properties files (server.properties, client.properties) and config files (server.config and client.config) in the C:\Borland\VisiBroker\examples\vbroker\security/corbaauthz folder.

The server configuration file is the JASS configuration file which defines the Hostlogin modules.

```
myrealm {
 com.borland.security.provider.authn.HostLoginModule required debug=true;
};
```

To enable security, you must set up certain porperties. Following are the properties set up in the server or client properties file:

Properties	Description
`vbroker.security.disable=false`	The default value is false. If set to `true`, disables all security services.
`vbroker.security.login=true`	IIf this property is set to `true`, during initialization, this property tries to log on to all the realms listed by property `vbroker.security.login.realms`.
`vbroker.security.login.realms=myrealm`	This gives a list of comma-separated realms to login to. This is used when login takes place, either through property `vbroker.security.login` (set to `true`) or API login.
`vbroker.security.peerAuthenticationMode=none`	Sets the peer authentication Mode. `when set to NONE`—Authentication is not required. During handshake, no certificate request will be sent to the peer. Regardless of whether the peer has certificates, a connection will be accepted. There will be no transport identity for the peer. For other authentication mode values, refer to the properties section "vbroker.security.peerAuthenticationMode"
`vbroker.security.domain.bank.defaultAccessRule=grant`	Specifies whether to grant or deny access to the domain by default in the absence of security roles for the provided domain. It handles `requests for methods not in the rolemap file`. Acceptable values are grant or deny.
`vbroker.security.authDomains=bank`	Specifies a comma-separated list of available authorization domains. For example: `vbroker.security.authDomains=domain1,domain2`

`vbroker.secuirty.logLevel=LEVEL_DEBUG`	Use this property to control the degree of logging.
`vbroker.security.login.realms=GSSUP#myrealm`	This gives a list of comma-separated realms to login to. This is used when login takes place, either through property `vbroker.security.login` (set to `true`) or API login.

Trust Providers

You can also plugin the assertion trust mechanism. Assertion can happen in a multi-hop scenario, or explicitly called through the assertion API. Server can have rules to determine whether the peer is trusted to make the assertion or not. The default implementation uses property setting to configure trusted peers on the server side. During runtime, peers must pass authentication and authorization in order to be trusted for making assertions. There can be only one Trust Provider for the entire security service.

To plugin the assertion trust mechanism, you will need to set the following properties:

```
vbroker.security.trust.trustProvider=MyProvider
vbroker.security.trust.trustProvider.MyProvider.property1=xxx
vbroker.security.trust.trustProvider.MyProvider.property2=xxx
```

The properties specified will be passed to the user plugin following the same mechanism as above.

7

Making Secure Connections (Java)

This section describes how to make secure connections for Java applications using VisiSecure. A brief introduction to the Java Secure Socket Extension (JSSE) is followed by the step-by-step details to securing an application.

JAAS and JSSE

VisiSecure uses the Java Authentication and Authorization Service (JAAS) to authenticate clients and servers to one another in J2EE applications. It provides a framework and standard interface for authentication users and assigning privileges. The VisiBroker Server uses the Java Secure Socket Extension (JSSE) to provide mechanisms for supporting SSL.

For information on the terms JAAS uses for its services, see “Basics of JAAS concepts”.

JSSE Basic Concepts

The VisiBroker ORB uses Internet Inter-ORB Protocol (IIOP) as its communication protocol. The Java Secure Socket Extension (JSSE) enables secure internet communications. It is a Java implementation of SSL and TLS protocols which include the functionality of data encryption, server and client authentication, and message integrity. JSSE also serves as a building block that can be simply and directly implemented in Java applications.

JSSE provides not only an API but also an implementation of that API. Implementations include socket classes, trust managers, key managers, SSLContexts, and a socket factory framework, in addition to public key certificate APIs.

JSSE also provides support for the underlying handshake mechanisms that are a part of SSL implementations. This includes cipher suite negotiation, client/server authentication, server session-management, and licensed code from RSA Data Security, Inc. JSSE uses Java KeyStores as a repository of Certificates and Private Keys. Further information on KeyStores can be obtained from Sun Microsystems’ JDK documentation. You can use JSSE properties for specifying trusted KeyStores and identity KeyStores.

Steps to secure clients and servers

Listed below are the common steps required for developing a secure client or secure server. For CORBA users the properties are all stored in files that are located through config files. Where ever appropriate the usage models for clients and servers are separately discussed. All properties can be set in the VisiBroker Management Console by right-clicking the node of interest in the Navigation Pane and selecting “Edit Properties.”

Note

These steps are similar for both Java and C++ applications.

Step One: Providing an identity

For authentication, you need username/password or certificates. Username/password and certificates can be collected from user through JAAS callback handlers. These can also be collected through LoginModules or through APIs.

For more information on server side and client side authentication, see “Authenticating clients with usernames and passwords”

For clients using usernames and passwords, there can be constraints about what the client knows about the server's realms.

For servers using username and password identities, authentication is performed locally since the realms are always known. There can be constraints on certificate identities as well, depending on whether they are stored in a KeyStore or whether they are specified through APIs. The KeyStore in VisiSecure for C++ refers to a directory structure similar to a `trustpointRepository`, which contains certificate chain.

Keeping these constraints in mind, the VisiSecure supports the following usage models: GSSUP based authentication and certificate based authentication.

You can use any of these to provide an identity to the server or client.

- “Username/password authentication, using JAAS modules, for known realms”
- “Username/password authentication, using APIs”
- “Certificate-based authentication, using KeyStores through property settings”
- “Certificate-based authentication, using KeyStores through APIs”
- “Certificate-based authentication, using APIs”
- “pkcs12-based authentication, using KeyStores”
- “pkcs12-based authentication, using APIs”

Username/password authentication, using JAAS modules, for known realms

If the realm to which the client wishes to authenticate is known, the client-side JAAS configuration would take the following form:

```
vbroker.security.login=true
vbroker.security.login.realms=<known-realm>
```

Username/password authentication, using APIs

The following code sample demonstrates the use of the login APIs. This case uses a wallet. For a full description of the four login modes supported, go to the VisiSecure for Java API and SPI sections.

```
public static void main(String[] args) {
   //initialize the ORB
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
   com.borland.security.Context ctx = (com.borland.security.Context)
```

```
        orb.resolve_initial_references("VBSecurityContext");
    if(ctx != null) {
        com.borland.securty.IdentityWallet wallet =
            new com.borland.security.IdentityWallet(<username>,
                <password>.toCharArray(), <realm>);
        ctx.login(wallet);
    }
}
```

Certificate-based authentication, using KeyStores through property settings

By setting the property `vbroker.security.login.realms=Certificate#ALL`, the client will be prompted for keystore location and access information. For valid values, see "Certificate mechanism".

Certificate-based authentication, using KeyStores through APIs

You can use the same APIs discussed in ""Username/password authentication, using APIs"" to log in using certificates through KeyStores. The realm name in the IdentityWallet should be `CERTIFICATE#ALL`. The `username` corresponds to an alias name in the default KeyStore that refers to a Key entry, and the `password` refers to the Private Key password (also the KeyStore password) corresponding to the same Key entry.

Certificate-based authentication, using APIs

If you do not want to use KeyStores directly, you can specify certificates and private keys using the `CertificateWallet`API. This class also supports the pkcs12 file format.

```
X509Certificate[] certChain = ...list-of-X509-certificates...
PrivateKey privKey = private-key
com.borland.security.CertificateWallet wallet =
    new com.borland.security.CertificateWallet(alias,
        certChain, privKey, "password".toCharArray());
```

The first argument in the new Certificate wallet is an alias to the entry in the KeyStore, if any. If you are not using keystores, set this argument to `null`.

pkcs12-based authentication, using KeyStores

You can use the same APIs discussed in "Username/password authentication, using APIs" to login using pkcs12 KeyStores. The realm name in the IdentityWallet should be `CERTIFICATE#ALL`, the `username` corresponds to an alias name in the default KeyStore that refers to a Key entry, and the `password` refers to the password needed to unlock the pkcs12 file. The property `javax.net.ssl.KeyStore` specifies the location of the pkcs12 file.

pkcs12-based authentication, using APIs

See "Certificate-based authentication, using APIs".

Step Two: Setting properties and Quality of Protection (QoP)

There are several properties that can be used to ensure connection with QoP. The VisiBroker ORB security properties for Java can be used to fine-tune connection quality. For example, you can set the `cipherList` property for SSL connections to set the cryptography strength.

QoP policies can be set using the `ServerQoPConfig` and the `ClientQoPConfig` APIs for servers and clients, respectively. These APIs allow you set target trust (whether or not targets must authenticate), the transport policy (whether or not to use SSL or another secure transport mechanism specified separately), and, for servers, an `AccessPolicyManager` that can access the RoleDB to set up access policies for POA objects. For QoP API information, see the VisiSecure for Java API and SPI section.

Step Three: Setting up Trust

Use the API `setTrustManager` for the proper security context to provide an `X509TrustManager` interface implementation. If you have certificates that need to be trusted, place them in a KeyStore and use `javax.net.ssl.trustStore` property to set it. A default `X509TrustManager` provided by the security service will be used if one is not provided.

Other trust policies are set in the QoP configurations. See "Step Two: Setting properties and Quality of Protection (QoP)".

Step Four: Setting up the Pseudo-Random Number Generator

Setting up the PRNG is required if you intend to use SSL communication.

- Construct a `SecureRandom` object and seed it.
- Set this object as your PRNG by using the `com.borland.security.Context` interface, `setSecureRandom` method.

For detailed information on the `com.borland.security.Context` interface, go to the VisiSecure for Java API and SPI book.

Step Five: If necessary, set up identity assertion

When a client invokes a method in a mid-tier server which, in the context of this request, invokes an end-tier server, then the identity of the client is internally asserted by the mid-tier server by default. Therefore, if `getCallerPrincipal` is called on the end-tier server, it will return the Client's principal.

Here the client's identity is asserted by the mid-tier server. The identity can be a username or a certificate. The client's private credentials such as private keys or passwords are not propagated on assertion. This implies that such an identity cannot be authenticated at the end-tier.

If the user would like to override the default identity assertion, there are APIs available to assert a given Principal. These APIs can be called only on mid-tier servers in the context of an invocation and with special permissions. For more information, see the VisiSecure for Java API and SPI section.

Security configuration while setting up a server engine

In order to use the secure transport while defining a custom server engine, the following properties need to be set for VisiSecure Java.

```
vbroker.se.<SE_NAME>.scms=<SSL_SCM_NAME>
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.manager.type=Socket
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.manager.ConnectionMax=0
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.manager.ConnectionMaxIdle=0
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.listener.type=SSL
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.listener.port=0
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.dispatcher.type=ThreadPool
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.dispatcher.threadMin=0
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.dispatcher.threadMax=0
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.dispatcher.threadMaxIdle=0
```

where:

`<SE_NAME>` is the name of the custom-defined server engine.

`<SSL_SCM_NAME>` is the name to be given to the SSL server connection manager (scm).

It is important to note that '*manager.type*' and '*listener.type*' must be initialized to the values indicated above whereas the rest of the properties can be set to any of the allowable values. For more information, see *Chapter 8* in Visibroker Developer's Guide. Here the properties are initialized to their defaults for illustration purpose.

Examining SSL related information

VisiSecure Server provides APIs to inspect and set SSL-related information. The `SecureContext` API is used to specify a Trust Manager, PRNG, inspect the SSL ciphersuites, and enable select ciphers.

Clients

To examine peer certificates, use the `getPeerSession()` to return an `SSLSession` object associated with the target. You can then use the standard JSSE APIs to obtain the information thereafter.

Servers

To examine peer certificates on the server side, you must set up the SSL connection with `com.borland.security.Context` and use the APIs with `com.borland.security.Current` to examine the `SSLSession` object associated with the thread.

SSL Example

The Bank SSL example included in the example folder of the visibroker directory contains a simple Bank interface to open a bank account and to query the balance. It illustrates basic communication using the ORB and SSL with VisiBroker for C++ and Java. In addition, this example demonstrates a modular approach to security by moving the code required to setup an SSL connection into initializers and properties.

From this example, you will learn how to:

- Request for secure transport in an application
- Install certificate identities in a server or a client
- Install a certificate in the trustpoint repository using the API or the property `vbroker.security.trustpointsRepository`
- Check the cipher suite and the identity of a peer
- Interoperate between C++ and Java

To run the example, ensure VisiBroker Smart Agent (osagent executable) is running on your network.

- Build the example in the directory by tying

`make -f Makefile_java` or

`nmake /f Makefile_java` on Windows

This will run the Bank.idl through the idl2java compiler. It will also build SecureServer.class, SecureClient.class and other class files.

This is the new and recommended way to add certificate-chain identity and to construct a CertificateWallet and log in with security context. The old method of inserting certificate chain, by resolving initial reference of "SecurityCurrent" on the orb, and using the API setPKprincipal using`(byte [][]derCertChain, byte[] privateKey, String passPhrase)`), still exists for backward compatibility.

For making the server run in the background, enter the following command:

```
prompt> vbj -DORBpropStorage=java_server.properties SecureServer
```

```
(start vbj -DORBpropStorage=java_server.properties SecureServer on Windows)
```

For making the C++ server run in the background, enter the following command:

```
prompt> SecureServer -DORBpropStorage=cpp_server.properties \ -
Dvbroker.orb.dynamicLibs="path to the dynamic library"/Init.so & (start SecureServer
...args... on Windows)
```

For connecting to java SecureClient,

```
prompt>vbj -DORBpropStorage=java_client.properties SecureClient
```

For connecting to C++ SecureClient

```
prompt>SecureClient -DORBpropStorage=cpp_client.properties
```

For setting up identity for the server, enter the commands below in the SecureServer

```
byte [][] certChain = {
  user_cert_1.getBytes (),
  user_cert_2.getBytes (),
  user_cert_3.getBytes (),
  user_cert_4.getBytes (),
      ca_cert.getBytes ()
};
```

To construct a CertificateWallet, enter the following commands in the SecureServer

```
com.borland.security.provider.CertificateWallet wallet =
          new com.borland.security.provider.CertificateWallet (null, certChain,
             encryptedPrivateKey.getBytes (), "Delt@$$$".toCharArray());
```

8

Making Secure Connections (C++)

This section describes how to make secure connections for C++ applications using VisiSecure.

Steps to secure clients and servers

Listed below are the common steps required for developing a secure client or secure server. For CORBA users the properties are all stored in files that are located through config files. Where ever appropriate the usage models for clients and servers are separately discussed. All properties can be set in the VisiBroker Management Console by right-clicking the node of interest in the Navigation Pane and selecting “Edit Properties.”

Note

These steps are similar for both Java and C++ applications.

Step One: Providing an identity

For authentication, you need username/password or certificates. Username/password and certificates can be collected from the user using JAAS callback handlers. These can also be collected through LoginModules or through APIs.

For more information on server side and client side authentication, see “Authenticating clients with usernames and passwords”

For clients using usernames and passwords, there can be constraints about what the client knows about the server's realms.

For servers using username and password identities, authentication is performed locally since the realms are always known. There can be constraints on certificate identities as well, depending on whether they are stored in a KeyStore or whether they are specified through APIs. The KeyStore in VisiSecure for C++ refers to a directory structure similar to a `trustpointRepository`, which contains certificate chain.

Keeping these constraints in mind, the VisiSecure supports the following usage models: GSSUP based authentication and certificate based authentication.

You can use any of these to provide an identity to the server or client.

- “Username/password authentication using LoginModules for known realms”

- "Username/password authentication using APIs"
- "Certificate-based authentication using KeyStores through property settings"
- "Certificate-based authentication using KeyStores through APIs"
- "Certificate-based authentication using APIs"
- "pkcs12-based authentication using KeyStores"
- "pkcs12-based authentication using APIs"

Username/password authentication using LoginModules for known realms

If the realm to which the client wishes to authenticate is known, the client-side configuration would take the following form:

```
vbroker.security.login=true
vbroker.security.login.realms=<known-realm>
```

Username/password authentication using APIs

The following code sample demonstrates the use of the login APIs. This case uses a wallet. For a full description of the four login modes supported, see "VisiSecure for C++ APIs" and "Security SPI for C++."

```
int main(int argc, char* const* argv) {
    // initialize the ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->resolve_initial_references("VBSecurityContext");
    Context* c = dynamic_cast<Context*> (obj.in());
    // Obtain a walletFactory
    CORBA::Object_var o = orb->resolve_initial_references("VBWalletFactory");
    vbsec::WalletFactory* wf = dynamic_cast<vbsec::WalletFactory*>(o.in());
    vbsec::Wallet* wallet = wf->createIdentityWallet( <username>, <password>, <realm>);
    c->login(*wallet);
}
```

Certificate-based authentication using KeyStores through property settings

By setting the property `vbroker.security.login.realms=Certificate#ALL`, the client will be prompted for the keystore location and the access information. For valid values, see "For Java:".

Certificate-based authentication using KeyStores through APIs

You can use the same APIs discussed in ""Username/password authentication using APIs"" to log in using certificates through KeyStores. The realm name in the IdentityWallet should be `CERTIFICATE#ALL`. The `username` corresponds to an alias name in the default KeyStore that refers to a Key entry, and the `password` refers to the Private Key password (also the KeyStore password) corresponding to the same Key entry.

Certificate-based authentication using APIs

If you do not want to use KeyStores directly, you can import certificates and private keys using the `CertificateFactory`API. This class also supports the pkcs12 file format.

```
CORBA::Object_var o = orb->resolve_initial_references("VBSecureSocketProvider");
vbsec::SecureSocketProvider* ssp = dynamic_cast<vbsec::SecureSocketProvider*>(o.in());

const vbsec::CertificateFactory& cf = ssp->getCertificateFactory ();
```

The first argument in the new Certificate wallet is an alias to the entry in the KeyStore, if any. If you are not using keystores, set this argument to `null`.

pkcs12-based authentication using KeyStores

You can use the same APIs discussed in "Username/password authentication using APIs" to log in using pkcs12 KeyStores. The realm name in the IdentityWallet should be `CERTIFICATE#ALL`, the `username` corresponds to an alias name in the default KeyStore

that refers to a Key entry, and the `password` refers to the password needed to unlock the pkcs12 file. The property `javax.net.ssl.KeyStore` specifies the location of the pkcs12 file.

pkcs12-based authentication using APIs

See "Certificate-based authentication using APIs".

Step Two: Setting properties and Quality of Protection (QoP)

There are several properties that can be used to ensure connection QoP. The VisiBroker ORB security properties for C++ can be used to fine-tune connection quality. For example, you can set the `cipherList` property for SSL connections to set up the cryptography strength.

QoP policies can be set using the `ServerQoPConfig` and the `ClientQoPConfig` APIs for servers and clients, respectively. These APIs allow you set target trust (whether or not targets must authenticate), the transport policy (whether or not to use SSL or another secure transport mechanism specified separately), and, for servers, an `AccessPolicyManager` that can access the RoleDB to set access policies for POA objects.

Step Three: Setting up Trust

Setting up of trust can be done through property `vbroker.security.trustpointRepository=Directory:<path to directory>`, where the directory contains the trusted certificates.

Other trust policies are set up in the QoP configurations. See "Step Two: Setting properties and Quality of Protection (QoP)".

Step Four: If necessary, set up identity assertion

When a client invokes a method in a mid-tier server which, in the context of this request, invokes an end-tier server, then the identity of the client is internally asserted by the mid-tier server by default. Therefore, if `getCallerSubject` is called on the end-tier server, it will return the Client's principal. Here the client's identity is asserted by the mid-tier server. The identity can be a username or certificate. The client's private credentials such as private keys ore passwords are not propagated on assertion. This implies that such an identity cannot be authenticated at the end-tier.

If the user would like to override the default identity assertion, there are APIs available to assert a given Principal. These APIs can be called only on mid-tier servers in the context of an invocation and with special permissions.

Security configuration while setting up a server engine

In order to be able to use secure transport while defining a custom server engine, the following properties needs to be set for VisiSecure for C++.

```
vbroker.se.<SE_NAME>.scms=<IIOP_SCM_NAME>,<SSL_SCM_NAME>
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.manager.type=Socket
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.listener.type=SSL
vbroker.se.<SE_NAME>.scm.<SSL_SCM_NAME>.dispatcher.type=ThreadPool
```

Where:

`<SE_NAME>` is the name of the custom-defined server engine.

`<SSL_SCM_NAME>` is the name to be given to the SSL server connection manager (scm).

`<IIOP_SCM_NAME>` is the name of an IIOP scm defined within the same sever engine.

It is important to note that the '*manager.type*' and the '*listener.type*' must be initialized to the values indicated above whereas the '*dispatcher.type*' can be any of the allowable types (For more information, see Chapter 8 in the Visibroker Developer's Guide. The default value is ThreadPool.

Another important point to be noted here is that a valid IIOP scm (<IIOP_SCM_NAME> in this case) must be defined *before* the SSL scm in the SCM list of custom server engine. All the properties of the IIOP scm can be set to any of the allowable values with the following two exceptions.

```
vbroker.se.<SE_NAME>.scm.<IIOP_SCM_NAME>.manager.type=Socket
vbroker.se.<SE_NAME>.scm.<IIOP_SCM_NAME>.listener.type=IIOP
```

For more information, see Chapter 8 in the VisiBroker Developer's Guide

Examining SSL related information

Borland VisiBroker provides APIs to inspect and set SSL-related information. The `SecureContext` API is used to inspect the SSL ciphersuites and enable select ciphers.

Clients

To examine peer certificates, use `getPeerSession()` to return an `SSLSession` object associated with the target. You can then use the standard JSSE APIs to obtain the information thereafter.

Servers

To examine peer certificates on the server side, you set up the SSL connection with `com.borland.security.Context` and use the APIs with `com.borland.security.Current` to examine the `SSLSession` object associated with the thread.

SSL example

This section demonstrates how to make a minimal SSL configuration for the client and server to communicate using SSL to enable them to perform mutual PKI authentication on the simplest, non-security aware VisiBroker example.

If you are using the same executables from basic/bank_agent to secure the non-security aware application, then no changes in the source codes are required.

Using properties to install certificates, private key and trustpoints

1. Copy only the executables of the Server (Server.exe on windows) and Client (Client.exe on windows) to this directory.
2. Make sure that osagent is up and running as usual
3. Launch the server using the command below:
   ```
   prompt> Server -DORBpropStorage=cpp_server.properties -
   Dvbroker.orb.dynamicLibs=<VisiSecure shared library name>
   ```
4. Launch the client using the command below:
   ```
   prompt> Client -DORBpropStorage=cpp_client.properties -
   Dvbroker.orb.dynamicLibs=<VisiSecure shared library name>
   ```
5. Open the property files **cpp_server.properties, cpp_client.properties** and notice how certificates and private keys are installed using wallet property set in that file.
6. Browse through the content of subdirectory **identities** and **trustpoints** and understand how the directory wallet and trustpoints are structured.

Note: VisiSecure shared library name> depends on the platforms.
For example:
on win32, it is vbsec.dll,
on Solaris 64 bit, it is vbsec64.so,
on HPUX 64 bit std build, it is vbsec64_p.sl.
It is recommended that you check your ${VBROKER_DIR}/lib directory

Using initializers to install certificates, private key, trustpoints and CRL

This section is similar to previous section, except that rather than using properties, shared libraries are used at runtime to install the certificates, keys, trustpoints and CRL. The advantage over the use of properties is that the shared library may also perform more security related initializations that are not possible to be done using properties, while at the same time, shared libraries keep the original non-secure aware application code intact. The shared libraries are transparent from the application point of view.

Build this example if you have not done so. In the previous section, we did not build, we just copied the executables. Building this example successfully, will create shared libraries: **ServerInit.<ext>** and **ClientInit.<ext>**.

Where **<ext>** depends on your platform.

for Windows, it is **dll**,

for Linux and Solaris, it is **so**,

for IBM, it is **a** and

for HP-UX, it is sl

1. Build the example by executing the command
   `nmake cpp` (for windows) or
   `make cpp` (for unix).
2. Make sure osagent is up and running.
3. Launch the server using the command below:

   ```
   prompt> Server -DORBpropStorage=cpp_server.properties -
   Dvbroker.orb.dynamicLibs=ServerInit.<ext>
   ```

4. Launch the client using the command below:

   ```
   prompt> Client -DORBpropStorage=cpp_server.properties -
   Dvbroker.orb.dynamicLibs=ClientInit.<ext>
   ```

5. Open the shared library source code `ServerInit.C` and `ClientInit.C` to notice how certificates, keys, trustpoints and CRL are installed on the ORB. The difference between `ServerInit.C` and `ClientInit.C` is only the set of certificates, keys, trustpoints and CRL that is installed. You can swap, for instance: ServerInit for client and ClientInit for server.
6. Reading through the code, you may notice that CRL will be installed only when we provide additional `-Dvbroker.app.useCRL` and therfore launching the server, for example, becomes,

   ```
   prompt> Server -DORBpropStorage=cpp_server.properties -Dvboker.app.useCRL=true -
   Dvbroker.orb.dynamicLibs=ServerInit.<ext>
   ```

Note: CRL, for this example, is prepared in such a way that when installed, the certificate that is directly issued by a trustpoint is revoked.

There is only one certificate directly issued by a trustpoint in this example. Therefore, any usage of CRL in any server or client or both will result in the SSL authentication failure and in turn, client will get NO_PERMISSION exception.

The failure can be because the client is not trusted by the server (if CRL is installed on the server) or vice versa (if the CRL is installed on the client) or both.

Using APIs with Security aware applications: SecureServer and SecureClient

This section demonstrates how the applications that are written with VisiSecure in mind take full advantage of control and power of VisiSecure features using API.

1. Build this example in this directory by executing the command:
   `nmake cpp (for windows) or`
   `make cpp (for unix).`

   When build successful, there will be executables created
   `SecureServer.exe on windows and SecureClient.exe on windows.`
2. Make sure the osagent is up and running.

3 Launch the server using the command below:
```
prompt> SecureServer
```

4 Launch the client using the command below:
```
prompt> SecureClient
```

5 Launch either the server or client or both using `-Dvbroker.app.useCRL=true,` and notice how the mutual SSL authentication fails and client gets `NO_PERMISSION` exception. For example,
```
prompt> SecureClient -Dvbroker.app.useCRL=true
```

6 Read and learn from `SecureServer.C, SecureClient.C`
- how they perform the security initialization in their `main()` and after `ORB_Init()`.
- how they impose `peerAuthenticationMode=require_and_trust` and `alwaysSecure=true` through QoP

Using APIs with pkcs12Server

This section demonstrates how to use VisiSecure API for handling of a PKCS12 storage, a very widely acceptable storage format for certificates and private keys.

1 Build this example in this directory by executing the command:
`nmake cpp` `(for windows) or`
`make cpp` `(for unix).`

When build successfully, there will be executables created `pkcs12Server.exe` on windows

1 Make sure the osagent is up and running

2 Launch the server using the command below:
```
prompt> pkcs12Server frans.pfx frans
```

3 Launch the client using the command below:
```
prompt> SecureClient
```

4 Launch the client using `-Dvbroker.app.useCRL=true`, and notice how the mutual SSL authentication fails and client gets `NO_PERMISSION` exception.
```
prompt> SecureClient -Dvbroker.app.useCRL=true
```

5 Read and learn from `pkcs12Server.C`
how it installs certificates and a private key from a PKCS12 file.

9

Security Properties for Java

Property	Description	Default
vbroker.security.logLevel	This property is used to control the degree of logging. 0 means no logging and 7 means maximum logging (debug messages).	0
vbroker.security.secureTransport	This property controls whether the transport connection is encrypted or not. If set to true, transport messages are encrypted. If set to false they are in the clear.	true
vbroker.security.alwaysSecure	This property together with the secureTransport property controls the default QoP on the client-side. If both are set to true, then the transport QoP is set to SECURE_ONLY; which means the client will only accept the secure transport. If either of them is set to false, then the client does not mandate security at the transport layer.	false
vbroker.security.server.transport	This property is used on the server side to define the server transport QoP. Acceptable values are CLEAR_ONLY, SECURE_ONLY or ALL. This allows the client that needs either CLEAR_ONLY or SECURE_ONLY to be able to connect to a server. This property will take effect only when the property secureTransport is true.	SECURE_ONLY
vbroker.security.disable	If this property is set to true, it disables all security services.	true
vbroker.security.transport.protocol	This property is used to select a security transport protocol. Possible values are SSL, SSLv2, SSLv3, TLS and TLSv1. For information on these protocols, see the Sun Microsystems documentation at: http://java.sun.com/products/jsse/doc/guide/API_users_guide.html#SSC.	TLSv1
vbroker.security.requireAuthentication	This server-side property is used to specify whether the client is required to authenticate or not	false
vbroker.security.authentication.callbackHandler	This property specifies the callback handler used for login modules used for interacting with users for credentials. You can specify one of the following or your own custom callback handler: com.borland.security.provider.authn.CmdLineCallbackHandler com.borland.security.provider.authn.HostCallbackHandler CmdLineCallbackHandler has password echo on, while HostCallbackHandler has password echo off.	n/a

Property	Description	Default
vbroker.security.cipherList	This property is used to set a list of comma-separated ciphers to be enabled by default on a startup. If not set, a default list of cipher suites will be enabled. These should be valid SSL Ciphers.	n/a
vbroker.security.authentication.config	This property specifies the path to the configuration file used for authentication.	null
vbroker.security.authentication.retryCount	This property specifies the number of retries count if remote authentication failed.	3
vbroker.security.authentication.clearCredentialsOnFailure	By default, if the authorization realm finds the authenticator to be incorrect after attaining the maximum number of retries, the ORB retains the authenticator. If you want the ORB to clear the authenticator (the credential) after the maximum number of retries, set this property to true.	false
vbroker.security.login	If set to true, at initialization-time, this property tries to log in to all the realms listed by property vbroker.security.login.realms.	false
vbroker.security.login.realms	This property gives a list of comma-separated realms to login to. This is used when the login takes place, either through property vbroker.security.login (set to true) or API login using login().	n/a
vbroker.security.vault	This property is used to specify the path to the vault file. This property will take effect regardless of whether vbroker.security.login is set to true or false.	n/a
vbroker.security.identity.reauthenticateOnFailure	When set to true, the security service will attempt reacquiring authentication information using the CallbackHandler. This property require the callback handler to be set either using the appropriate property or at runtime by calling the appropriate method.	false
vbroker.security.identity.enableReactiveLogin	When set to true, the security service behaves as follows: If the security service cannot find an identity for any of the targets supported by a server it is attempting to communicate with, it will then attempt to acquire credentials for one of the targets in the target object's IOR. If a corresponding authentication realm is available for this target (that the user chooses to provide credentials for), then authentication is also attempted locally. The Reactive login requires a callback handler to be set either using the appropriate property or at runtime by calling the appropriate method.	true
vbroker.security.authDomains	This property specifies a comma-separated list of available authorization domains. For example: vbroker.security.authDomains=<dom1>,<doma2>…	null
vbroker.security.domain.<domain_name>.rolemap_path	This property specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain <domain_name> specified in vbroker.security.authDomains.	n/a
vbroker.security.domain.<domain_name>.rolemap_enableRefresh	When set to true, this property enables dynamic loading of the RoleDB file specified in vbroker.security.domain.<domain_name>.rolemap_path property. The interval of dynamic loading is specified by property vbroker.security.domain.<domain_name>.rolemap_refreshTimeInSeconds.	false
vbroker.security.domain.<domain_name>.rolemap_refreshTimeInSeconds	This property specifies the rolemap refresh time in seconds.	300
vbroker.security.domain.<domain name>.runas.<run_as_role_name>	This property specifies the name of the run-as role. The value can be either use-caller-identity to have the caller principal be in the run-as role, or specify an alias for a run-as principal for the run-as role name.	n/a

Property	Description	Default
vbroker.security.peerAuthenticationMode	This property sets the peer authentication Mode. The possible values are: REQUIRE REQUIRE_AND_TRUST REQUEST REQUEST_AND_TRUST NONE Note that the REQUEST and REQUEST_AND_TRUST modes cannot receive peer certificate chains due to JSSE restrictions.	NONE
vbroker.security.trustpointsRepository	This propery specifies a path to the directory containing trusted certificates and CRLs or to a trusted Keystore whose values are implementations of TrustedCertificateEntry. Default values are either a directory, given in the format Directory:<path_to_certs> or a Keystore, given in the format Keystore:<path_to_keystore>.	n/a
vbroker.security.defaultJSSETrust	If this property is set to true, the JSSE default trust files like cacerts and jssecacerts, if present in JRE, it will be used to load trusted certificates.	false
vbroker.security.assertions.trust.<n>	This property is used to specify a list of trusted roles (specified with the format <role>@<authorization_domain>.) <n> is a uniquely identified for each trust assertion rule as a list of digits. For example, setting vbroker.security.assertions.trust.1=ServerAdmin@default means this process trusts any assertions made by the ServerAdmin role in the default authorization domain.	n/a
vbroker.security.assertions.trust.all	Setting this property to true will trust all assertions made by peers.	false
vbroker.security.server.requireUPIdentity	Set this property to true, if the server requires the client to send a Username/Password for authentication (regardless of certificate-based authentication). This is a server-sided property.	n/a
vbroker.security.cipherList	Using this property, you can set a list of comma-separated ciphers to be enabled by default on startup. If not set, a default list of ciphersuites will be enabled. These should be valid SSL Ciphers.	n/a
vbroker.security.controlAdminAccess	Set this property to true for enabling Server Manager operations on a Secure Server.	false
vbroker.security.serverManager.authDomain	This property points to a security domain listed in vbroker.security.authDomains. The specified domain is used for the Server Manager's role-based access control checks. A rolemap must be specified for the domain.	n/a
vbroker.security.serverManager.role.all	This property specifies the role name required for accessing all Server Manager operations.	n/a
vbroker.security.serverManager.role.<method_name>	This property specifies the role name required for accessing the specified method of the Server Manager.	n/a
vbroker.security.domain.<domain_name>.defaultAccessRule	This property specifies whether to grant or deny access to the domain by default in the absence of security roles for the provided domain. Acceptable values are grant or deny.	grant
vbroker.se.iiop_tp.scm.ssl.listener.trustInClient	This is a server-sided property. Set this property to true to have the server require certificates from the client. These certificates must also be trusted by the server by setting the appropriate server-side trust properties. For more information, see the vbroker.security.trustpointsRepository property and the vbroker.security.defaultJSSETrust property.	false
vbroker.security.wallet.type	A wallet is a set of directories containing encrypted private keys and certificate chains for each identity. Use this property to point to the directory containing the directories for all identities, using the format: Directory:<path_to_identities>	n/a

Property	Description	Default
vbroker.security.wallet.identity	This property is used to point to a directory within the path defined in vbroker.security.wallet.type that contains keys and/or certificate information for a specific identity. Note that the value of this property must consist of lower-case letters only.	n/a
vbroker.security.wallet.password	This property specifies the password used to decrypt the private key or the password associated with the login.	n/a
vbroker.security.TSS.authenticationTimeToLive	This property sets the time for reauthentication.	600 sec
vbroker.security.TSS.stateful	This coresponds to the IOR component CompoundSecMechList field 'stateful' (see OMG specs). This signifies whether or not the server supports 'stateful' SAS session and therefore the client can make decisions according to the standard behaviour as defined by the OMG specifications.	true
vbroker.security.trustProvider	This property is a part of the trust provider plugability mechanism: vbroker.security.trust.trustProvider=xyz <== where xyz can be any string. The FQCN of the trustprovider class implementation: vbroker.security.trustProvider.xyz.provider=<FQCN of the trustprovider class impl> You can set properties specific to the trustprovider xyz: vbroker.security.trustProvider.xyz.property1=value1 <== vbroker.security.trustProvider.xyz.property2=value2 <==	set any string
vbroker.security.supportIdentityAssertion	This property corresponds to the IdentityAssertion flag in the IOR sub-component SASContextSec (See the OMG specifications) The default value is true. When set to true, it will set the coresponding bit in the component. When set to false, it will reset it. This bit signifies whether or not the server supports identity assertion and therfore, the client can react according to the pre-defined behaviour associated with this bit. (See OMG specifications).	true
vbroker.security.rolemap_path	This property specifies the location of the RoleDB file that describes the roles used for the authorization. This is scoped within the domain <domain_name> specified in: vbroker.security.authDomains.	n/a
vbroker.security.authentication.callbackHandler=com.borland.security.provider.authn.DialogCallbackHandler	This property when set to com.borland.security.provider.authn.DialogCallbackHandler enables to pop out a GUI Login.	com.borland.security.provider.authn.DialogCallbackHandler

10

Security Properties for C++

Property	Description	Default
vbroker.security.logLevel	This property is used to control the degree of logging. Acceptable values are: LEVEL_WARN, LEVEL_NOTICE, LEVEL_INFO, and LEVEL_DEBUG strings.	LEVEL_WARN
vbroker.security.logFile	This property is used to redirect the log output to a file. The default log output is to std::cerr.	null
vbroker.security.secureTransport	This property determines whether the secure transport is supported or not. If set to false, transport uses CLEAR_ONLY. This property also determines if the client side of the ORB is always connected to the server using SSl. In cases, when server does not support SSL, it will not connect.	true
vbroker.security.alwaysSecure	This is a client-side only property. It determines whether to use secure transport only or not. **Note:** To use secure transport only, the secureTransport property must also be set to true.	true
vbroker.security.server.transport	This is a server-side only property. It defines whether the server transport is: CLEAR_ONLY, SECURE_ONLY or ALL. This property will not take effect when the secureTransport property is set to false.	SECURE_ONLY
vbroker.security.disable	If this property is set to true, it disables all security services.	false
vbroker.security. requireAuthentication	A server-side only property. It is used to specify whether the client is required to authenticate.	true
vbroker.security.authentication. callbackHandler	Thid property specifies the callback handler for login modules to use for interacting with the user for credentials. You can specify one of the following or your own custom callback handler. For more information, see "VisiSecure for C++ APIs." com.borland.security.provider.authn.CmdLineCallbackHandler com.borland.security.provider.authn.HostCallbackHandler CmdLineCallbackHandler has password echo on, while HostCallbackHandler has password echo off.	HostCallbackHandler
vbroker.security.authentication. config	This property specifies the path to the configuration file used for authentication.	null
vbroker.security.authentication. retryCount	This property specifies the number of retries if the remote authentication failed.	3

Property	Description	Default
vbroker.security.login	If set to true at initialization-time, this property tries to login to all the realms listed by property: vbroker.security.login.realms.	true
vbroker.security.login.realms	This gives a list of comma-separated realms to login to. This is used when login takes place, either through property vbroker.security.login (set to true) or API login.	n/a
vbroker.security.vault	This property is used to specify the path to the vault file. This property will take effect regardless of whether vbroker.security.login is set to true or false.	n/a
vbroker.security.identity.reactiveLogin	When set to true, the security service behaves as follows. If the security service cannot find an identity for any of the targets supported by a server it is attempting to communicate with, it then attempts to acquire credentials for one of the targets in the target object's IOR. If a corresponding authentication realm is available for this target (that the user chooses to provide credentials for), then authentication is also attempted locally. Reactive login requires a callback handler to be set either using the appropriate property or at runtime by calling the appropriate method. The default handler is HostCallbackHandler.	true
vbroker.security.authDomains	This property specifies a comma-separated list of available authorization domains. For example: vbroker.security.authDomains=domain1,domain2	n/a
vbroker.security.domain.<domain-name>.rolemap_path	This property specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain <domain_name> specified in: vbroker.security.authDomains.	n/a
vbroker.security.domain.<domain_name>.rolemap_enableRefresh	When set to true, this property enables dynamic loading of the RoleDB file specified in vbroker.security.domain.<domain_name>.rolemap_path property. The interval of dynamic loading is specified by the property vbroker.security.domain.<domain_name>.rolemap_refreshTimeInSeconds.	false
vbroker.security.domain.<domain_name>.rolemap_refreshTimeInSeconds	This property specifies the rolemap refresh time in seconds.	300
vbroker.security.domain.<domain_name>.defaultAccessRule	When clients are just about to access CORBA resources, access control decides whether to allow or disallow. Decisions are made based on what roles are required to access the resources and compared against the fact whether the client exists atleast in one of the roles. Unfortunately, there are situations, in which system fails to know the roles required to access some resource, the value of the above property [grant/deny] will determine the decision being made in this kind of situation.	grant/deny
vbroker.security.cert.basicConstraintCritical	As per the new X509 V3 standard, non end-user certificates in a certificate chain must have extensions that is called “basic constraint”. According to the standard, when present, this extension must be marked as “critical” enforcing the recipient to “must understand” and process accordingly. As the later enforcement is too strict in respect to the earlier implementation that is aware only of X509 V1.	True/false. The default is false.
vbroker.security.config.root	This is an absolute path of the directory, in respect to which all relative file system references are being made for various VisiSecure config files.	
vbroker.security.server.requireUPIdentity	The server side on this ORB will require incoming requests to carry valid CSIV2 service context based identity.	
vbroker.security.trust		
vbroker.security.identityAssertion=[true/false]	The server side of this ORB is enabled to propogate the IdentityTokem sent as part of CSIV2 service context to the next tier (if any)	

Property	Description	Default
vbroker.security.peerAuthenticationMode	This property sets the peer authentication Mode. Possible values are: REQUIRE—Peer certificates are required to establish a connection. If the peer does not present its certificates, the connection will be refused. Peer certificates will also be authenticated, if not valid, the connection will be refused. If required, transport identity can be established using these certificates. In this mode, peer certificates are not required to be trusted. REQUIRE_AND_TRUST—Same as REQUIRE mode, except that the peer certificates need to be trusted, otherwise the connection will be refused. REQUEST—Peer certificates will be requested. The peer is not required to have certificates; no transport identity will be established when peer does not have certificates. However, if a peer does present certificates, the certificates will be authenticated; if not valid, the connection will be refused. If required, transport identity can be established using these certificates. In this mode, peer certificates are not required to be trusted. REQUEST_AND_TRUST—Same as REQUEST mode except that the peer certificates need to be trusted, otherwise the connection will be refused. NONE—Authentication is not required. During handshake, no certificate request will be sent to the peer. Regardless of whether the peer has certificates, a connection will be accepted. There will be no transport identity for the peer.	REQUIRE_AND_TRUST
vbroker.security.trustpointsRepository	Specifies a path to the directory containing trusted certificates. These are given in the form Directory:<certs_dir>. For example: `vbroker.security.trustpointsRepository=Directory:c:\data\identities\Delta`	n/a
vbroker.security.assertions.trust.<n>	This property is used to specify a list of trusted roles. (specified within the format <role>@<authorization_domain>). <n> is uniquely identified for each trust assertion rule as a list of digits. For example, setting vbroker.security.assertions.trust.1=ServerAdmin@default means this process trusts any assertion made by the ServerAdmin role in the default authorization domain.	n/a
vbroker.security.assertions.trust.all	Setting this property to true will trust all the assertion made by peers.	false
vbroker.security.server.requireUPIdentity	A server-side only property. If the server requires the client to send a username/password for authentication (regardless of certificate-based authentication), set it to true. If vbroker.security.login.realms is set, then this property is automatically set to true. However, you can override it by explicitly setting it in the property file.	n/a
vbroker.security.cipherList	This property can be set to a list of comma-separated ciphers to be enabled by default on startup. If not set, a default list of cipher suites will be enabled. These should be valid SSL Ciphers.	n/a
vbroker.security.wallet.type	A wallet is a set of directories containing encrypted private keys and certificate chains for each identity. Use this property to point to the directory containing the directories for all identities, using the format: Directory:<path_to_identities>	n/a
vbroker.security.wallet.identity	This property points to a directory within the path defined in vbroker.security.wallet.type that contains keys and/or certificate information for a specific identity.	n/a
vbroker.security.wallet.password	This property specifies the password used to decrypt the private key or the password associated with the login.	n/a

32

VisiSecure for C++ APIs

This section describes APIs that are defined in the VisiSecure for C++. It is separated into subsections including:

- General APIs
- SSL and Certificate APIs
- QoP APIs
- Authorization APIs

All classes are under the namespace `vbsec` unless otherwise specified.

General API

The general VisiSecure API describes the `Current` and `Context` APIs. It provides the API information for Principals, Credentials, and Subjects. In addition, the `Wallet` API is discussed.

class vbsec::Current

Current represents the view to the thread specific security context. Some calls are relevant only in an request execution context. This object can be obtained through the following code:

```
CORBA::Object_var obj = orb-
>resolve_initial_references("VBSecurityCurrent");
Current* c = dynamic_cast(obj.in());
```

Include File

The `vbsec.h` file should be included when you use this class.

Methods

void asserting (const vbsec::Subject* caller)

Assert a subject as caller identity.

Parameter	Description
caller	The caller name of the subject.

void clearAssertion ()

Clear an assertion made by any previous API call of `asserting`. The caller before the assertion is made will be restored as the caller for next invocation. This API shall be used in conjunction with `asserting`. Mismatching calls of these two methods may cause undesired caller identities or unexpected exceptions.

const vbsec::Subject* getPeerSubject ()

Accesses the peer subject.

Returns

The pointer to a Subject object representing the peer.

const vbsec::Subject* getCallerSubject ()

Accesses the caller subject.

Returns

The pointer to a Subject object representing the caller.

const vbsec::SSLSession* getPeerSession (CORBA::Object* peer)

Get the peer `SSLSession`. This call returns the `SSLSession` of the client peer for this request. This method cannot be called outside the context of a request.

Parameter	Description
peer	A peer object retrieved from the bind.

Returns

The pointer to a SSLSession currently established.

Exceptions

`BAD_OPERATION` is thrown if this method is called outside the context of a request or when called in a request context where the request was received over a clear TCP connection.

class vbsec::Context

`Context` represents the security context under which a client will execute. This class can be obtained through the following code:

```
CORBA::Object_var obj = orb-
>resolve_initial_references("VBSecurityContext");
Context* c = dynamic_cast(obj.in());
```

Include File

The `vbsec.h` file should be included when you use this class.

Methods

```
void login()
```

Login into the system. This logs-in to the realms defined in the property `vbroker.security.loginRealms`. It traverses the list of realms specified and authenticates against each realm.

void login (vbsec::CallbackHandler& handler)

Use this to login to the system using the specified `CallbackHandler` to obtain the login information.

Parameter	Description
handler	The default callback handler to be use for acquiring information.

void login (const std::string& realm)

Use this to login into the system for a specific realm.

Parameter	Description
realm	The realm to login to.

void login (const std::string& realm, vbsec::CallbackHandler& handler)

Use this to login into the system for a given realm, using a given callback handler for acquiring information.

Parameter	Description
realm	The realm to login to.
handler	The default callback handler to be use for acquiring information.

void login (const vbsec::Wallet& wallet)

Use this login into the system with a `wallet`. Wallet can be created using `WalletFactory` API.

Parameter	Description
wallet	The wallet to be used for login.

void login (const std::vector<const vbsec::Wallet*>& wallet)

Use this to login into the system with a set of wallets specified as a vector.

Parameter	Description
wallet	A wallet to be used for login

const vbsec::Subject* getSubject (const std::string& realm)

Gets the `Subject` corresponding to a given realm.

Parameter	Description
realm	The Realm for the Principal

Returns

A pointer to the `Subject` object representing the subject of the realm.

void loadVault (std::istream& stream, const CSI::UTF8String& vaultPass)

Loads a given vault. The identities in the vault are loaded into the system. No login required when this method is used.

Parameter	Description
stream	Stream that the vault information will be read from, in binary format.
vaultPass	Password used to decrypt the vault information.

void logout()

Log the user out from all the realms.

void logout (const std::string& realm)

Log the user out from a given realm.

Parameter	Description
realm	The realm to logout from.

void setCallbackHandler (vbsec::CallbackHandler* handler)

Set the default callback handler programmatically. This is similar to using the property `vbroker.security.authentication.callbackHandler`.

Parameter	Description
handler	The `CallbackHandler` to be set.

void generateVault(std::ostream& stream, const CSI::UTF8String& password)

This generates a vault. The vault is written out to the stream that is passed in and encrypted using the password provided (also used to decrypt the vault). The password may be null. The vault contains all of the system's identities.

Parameter	Description
stream	The stream that the vault information will be written into, in binary format.
password	The password used to encrypt the vault information.

vbsec::Subject* authenticateUser (const vbsec::Wallet& wallet)

This authenticates the given wallet credential. The login will be performed using the wallet but the authenticated subject will not be used as one of the system identities.

Parameter	Description
wallet	The wallet to be used for authentication

vbsec::Subject* importIdentity (const vbsec::Wallet& wallet)

Import a subject using the given wallet credential. No login is required with this method. The subject will not be used as one of the system identities.

Parameter	Description
wallet	The wallet corresponding to the identity to be imported.

void setPRNGSeed (const CORBA::OctetSequence& seed)

This sets a seed for the pseudo-random generator used by the SSL layer.

Parameter	Description
seed	The seed for the PRNG.

ssl::CipherSuiteInfoList* listAvailableCipherSuites()

This gets the list of cipher suites that are available for use with the SSL layer. Note that this is different from the `getEnabledCipherSuites` call in that not all the **available** cipher suites may be currently enabled.

Returns

List of cipher suits that are available but may not be enabled for use with the SSL layer.

void enableCipherSuites (const ssl::CipherSuiteInfoList& suites)

This sets the cipher suites that should be enabled for all SSL sessions.

Parameter	Description
suites	An IDL-generated `CipherSuiteInfoList` type.

ssl::CipherSuiteInfoList* getEnabledCipherSuites()

Gets the set of cipher suites that are currently enabled for all SSL sessions.

Returns

the Cipher suits that are currently enabled for all SSL sessions.

void setSSLContext (vbsec::VBSSLContext* ctx)

This sets the SSL context. This will allow establishing of an SSL session using the information defined in `VBSSLContext`. A `VBSSLContext` can be created using the `SecureSocketProvider` API.

Parameter	Description
ctx	The `VBSSLContext` that is to be used for any SSL session establishment.

VBSSLContext& getSSLContext()

Get the `VBSSLContext` that is set using the `setSSLContext()` or return a default `VBSSLContext` object.

Returns

The `VBSSLContext` that will be used for any `SSLSession` establishment.

class vbsec::Principal

The Principal represents the identity of a user. This is a virtual class.

Include file

The `vbsec.h` file should be included when you use this class.

Methods

std::string getName() const

Returns

The name of the Principal.

`std::string toString() const`

Get the string representation of the Principal.

Returns

The string representation of the Principal.

class vbsec::Credential

Credential represents the information used to authenticate an identity, such as user name and password. This is a virtual class.

Include File

The `vbsec.h` file should be included when you use this class.

class vbsec::Subject

The Subject represents a grouping of related information for a single entity, such as a person. Such information includes the Subject's identities as well as its security-related attributes (passwords and cryptographic keys, for example).

Include File

The `vbsec.h` file should be included when you use this class.

Methods

Principal::set& getPrincipals()

Gets the principals in the subject.

Returns

The set of the principals in the subject. Modifying the content of the set will have no effect on the subject.

void clearPrincipals()

Clears the principals from the subject. All principals in the subject are removed.

Credential::set& getPublicCredentials()

Get the public credentials in the subject—public keys for example.

Returns

The set of the public credential in the subject. Modifying the content of the set will have no effect on the subject.

void clearPublicCredentials()

Clear the public credentials in the subject. All public credentials in the subject will be destroyed and removed.

Credential::set& getPrivateCredentials()

Get the private credentials in the subject—private keys for example.

Returns

The set of the private credential in the subject. Modifying the content of the set will have no effect on the subject.

void clearPrivateCredentials()

Clear the private credentials in the subject. All private credentials in the subject will be destroyed and removed.

Principal::set getPrincipals (const type_info& info) const

Gets a set of principals in the subject which have the same runtime type information as provided.

Parameter	Description
info	The runtime type information that the returned principals shall have.

Returns

A set of the principals in the subject which have same runtime information as the given one. Modifying the content of the set will have no effect on the subject.

Credential::set getPublicCredentials (const type_info& info) const

Get set of public credentials in the subject which have the same runtime type information as provided.

Parameter	Description
info	The runtime type information that the returned public credential shall have.

Returns

A set of the public credential in the subject which have same runtime information as the given one. Modifying the content of the set will have no effect on the subject.

Credential::set getPrivateCredentials (const type_info& info) const

Get set of private credentials in the subject which have the same runtime type information as provided.

Parameter	Description
info	The runtime type information that the returned private credentials shall have.

Returns

A set of the private credentials in the subject which have same runtime information as the given one. Modifying the content of the set will have no effect on the subject.

class vbsec::Wallet

A `Wallet` is a holder of credentials usually used in login API calls. A `Wallet` can be created using `WalletFactory` APIs and contain multiple types of credentials.

Include File

The `vbsec.h` file should be included when you use this class.

Methods

`std::string getTarget () const`

Get the target to which wallet authenticates.

Returns

The string representation of the target information.

`void populateSubject (Subject& subject)`

Populate the given subject with necessary credentials or other information for authentication.

Parameter	Description
subject	The subject for the wallet to populate.

class vbsec::WalletFactory

WalletFactory is a factory class to create multiple types of wallets.

Include File

The `vbsec.h` file should be included when you use this class.

Methods

Wallet* createCertificateWallet (const std::string& name,
const std::string& password,
const std::string& alias,
const std::string& keypassword,
short usage)

Create a certificate wallet using a C++ keystore. The C++ keystore is similar to the Java keystore but is implemented using a directory structure. When logging in with a wallet created by this API, the certificate chain will be used in the SSL layer.

Parameter	Description
name	The directory name of the keystore.
password	The password for the keystore, **not used for this release**.
alias	The alias to be used in the keystore.
keypassword	The password for the private key of the given alias.
short usage	The usage of the certificate information, `CLIENT`, `SERVER` or `ALL`.

Returns

Certificate wallet that contains the given information.

Wallet* createCertificateWallet (const CORBAsec::X509CertList& chain,
const CORBAsec::ASN1Object& privkey,
const CSI::UTF8String& password)

Create a certificate wallet using a certificate chain, private key and password.

Parameter	Description
chain	The certificate chain to create the wallet.
privkey	The private key of the certificate chain.
password	The password for the private key.

Returns

Certificate wallet that contains the given information.

Wallet* createIdentityWallet (const std::string& username,
const std::string& password,
const std::string& realm)

Create a identity wallet using a username, password and realm that the wallet to which the wallet authenticates.

Parameter	Description
username	The username of the identity.
password	The password for the identity.
realm	The realm to which the wallet authenticates.

Returns

Identity wallet that contains the given information.

Wallet* createIdentityWallet (const std::string& username,
const std::string& password,
const std::string& realm,
const std::vector<std::string>& groups)

Create a identity wallet using a username, password, realm to which the wallet authenticates, and a set of group attributes.

Parameter	Description
username	The username of the identity.
password	The password for the identity.
realm	The realm to which the wallet authenticates.
groups	A set of group attributes to which the identity belongs.

Returns

Identity wallet that contains the given information.

SSL API

This section explains the various SSL APIs that interact with VisiSecure's SSL implementation.

class vbsec::SSLSession

`SSLSession` represents the session of the current SSL connection. The `SSLSession` can be gotten from `vbsec::Context using getPeerSession()`.

Include File

The `vbssp.h` file should be included when you use this class.

Methods

time_t getEstablishmentTime() const

Get the time when the SSL connection was established.

Returns

The time when the SSL connection was established.

const ssl::CipherSuiteInfo& getNegotiatedCipher() const

This method returns the negotiated cipher from the peer for a given SSL connection.

Returns

The negotiated cipher from the peer for a given SSL connection.

```
const CORBAsec::X509CertList& getPeerCertificates() const
```

Get the certificate chain of the peer.

Returns

Peer certificate chain.

const CORBAsec::X509Cert* getTrustpoint() const

Get the trust point by which the peer is trusted. Null will be returned if peer does not have certificates or its certificates are not trusted.

Returns

The trust point by which the peer is trusted, or null if not.

char* getPeerAddress() const

Get the IP address of the peer.

Returns

Peer IP address in a string with the following format: xxx.xx.xx.xx.

CORBA::UShort getPeerPort() const

Returns the peer port number used by this connection.

Returns

The port number of the peer on the connection.

void prettyPrint (std::ostream& os) const

Print the SSLSession information into the given output stream.

Parameter	Description
os	The output stream to print the SSLSession information.

class vbsec::VBSSLContext

`VBSSLContext` contains information needed to establish an `SSLSession`. This object is created using `SecureSocketProvider::createSSLContext()`.

Include File

The `vbssp.h` file should be included when you use this class.

Methods

const CORBAsec::X509CertList& getCertificates() const

Get the certificate chain representing the identity to be used for the SSL layer.

Returns

The certificate chain representing the identity to be used for the SSL layer.

```
void setCipherSuiteList (const ssl::CipherSuiteInfoList& list)
```

This method is used to specify the ciphers available for the SSL connections.

Parameter	Description
list	A list of ciphers that should be available for the SSL connections.

const ssl::CipherSuiteInfoList& getCipherSuiteList() const

Return the ciphers that are currently used by the SSL layer.

Returns

The ciphers that are currently used by the SSL layer.

void addTrustedCertificate (const CORBAsec::X509Cert& trusted,const CORBAsec::ASN1Object* crl = NULL)

Programmatically add trusted certificate into the SSL context.

Parameter	Description
trusted	Certificate that is to be trusted.
crl	The CRL issued by the trusted certificate default value of NULL means no CRL Applications can call this method passing only the first argument, in which case the default value of NULL applies.

CORBAsec::X509CertList* getTrustedCertificates() const

Get list of certificates that are trusted.

Returns

List of certificates that are trusted.

class ssl::CipherSuiteInfo

CipherSuiteInfo is a structure containing two fields:

- `CORBA::ULong SuiteID`
- `CORBA::String_var Name`

This IDL structure contains two fields which describe ciphers according to the SSL specification. The list of `SuiteID` values and their names is in the include file, `ssl_c.h`.

Include File

The `ssl_c.hh` file should be included when you use this class.

class CipherSuiteName

This class provides information about the ciphers used in the Security Service.

Include File

The `csstring.h` file should be included when you use this class.

Methods

static const char* toString (int tag)

Return a standard representation of a supported SSL cipher.

Parameter	Description
tag	tag associated with the cipher name.

Returns

Returns a stringified description of the cipher.

static const int fromString (char* description)

Give the tag associated to the given cipher description.

Parameter	Description
description	The stringified description of the cipher.

Returns

The tag associated with the cipher name provided as the argument.

class vbsec::SecureSocketProvider

A `SecureSocketProvider` is the provider for secure socket connections. It provides the function of creating the SSL context, handling SSL certificates, and managing other secure socket-related information.

Include File

The `vbssp.h` file should be included when you use this class.

Methods

vbsec::VBSSLContext* createSSLContext (const CORBAsec::X509CertList& chain, const CORBAsec::ASN1Object& privkey, const CSI::UTF8String& password)

This method create a SSL context using the given information. The SSL context can then be passed into `vbsec::Context` and used to establish an SSL connection.

Parameter	Description
chain	The certificate chain
privkey	The private key object.
password	The password for the private key.

Returns

`VBSSLcontext` containing the given information.

void setPRNGSeed (const ssl::Current::PRNGSeed& seed)

Sets a seed for the pseudo-random number generator used by the SSL layer.

Parameter	Description
seed	The seed for the PRNG.

const ssl::CipherSuiteInfoList& listAvailableCipherSuites() const

Gets the list of cipher suites that are available for use with the SSL layer. Note that this is different from the `getEnabledCipherSuites` call in that not all the available cipher suites may be currently enabled.

Returns

List of cipher suits that are available but may not be enabled for use with the SSL layer.

const CertificateFactory& getCertificateFactory() const

Gets a certificate Factory.

Returns

A `CertificateFactory` object.

class ssl::Current

The `ssl::Current` lets your client application or server object set its private key and offer its certificate information to its peer. This interface also lets you configure the SSL connection and associate your certificates and private key with an SSL connection.

Be aware that private keys and certificates contain header and trailer lines, which mark the beginning and end of the key or certificate. All of the methods offered by this interface for setting private keys and certificate chains require that these header and trailer lines be present. The parsing rules for these lines is:

- The recognized header line format for certificates is:

```
-----BEGIN CERTIFICATE-----
```

- The recognized header line format for private keys is:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
```

- All header lines must end with a new line character.

– All trailer lines must be preceded with, and end with, a newline character. PEM-style private keys have two additional header lines that other private keys do not have: Proc-Type and DEK-Info. Both of these lines must be present and they must end with new line characters.

This object can be obtained through the following code:

```
CORBA::Object_var obj = orb->resolve_initial_references("SSLCurrent");
ssl::Current_var current = ssl::Current::_narrow(obj);
```

Include File

The `ssl_c.hh` file should be included when you use this class.

Methods

CORBA::ULong getNegotiatedCipher(CORBA::Object_ptr peer)

This method returns the negotiated cipher from the peer for a given SSL connection.

Parameter	Description
peer	The peer from which you obtain the negotiated cipher.

Returns

A value (tag) representing the cipher used. (Use `CipherSuiteName::toString` to get a String representation.)

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL.

CORBAsec::X509CertList_ptr getPeerCertificateChain(CORBA::Object_ptr peer)

This method obtains the peer's certificate chain. It is usually invoked by a client application to obtain information from a server, but a server can optionally request information from a client.

Parameter	Description
peer	The peer from which you obtain the negotiated cipher.

Returns

A value representing the cipher used. (Use CipherSuiteName::toString to get a String representation.)

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL.

char* getPeerAddress(CORBA::Object_ptr peer)

Returns a description of the socket parameters used by this connection.

Parameter	Description
peer	The peer from which you obtain the information.

Returns

Peer IP address in a string with the following format: xxx.xx.xx.xx

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL.

CORBA::Boolean isPeerTrusted(CORBA::Object_ptr peer)

Tests if the certificate chain of the peer is trusted—that is, if one certificate of the chain is in the trustpoint.

Parameter	Description
peer	The peer from which you obtain the information.

Returns

`true` if the chain is trusted, `false` otherwise.

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL.

trust::Trustpoints_ptr Trustpoints getTrustpointsObject()

Returns a reference to the trustpoint repository. Use this API to access trustpoints object and set trustpoints.

Returns

A reference to the trustpoint repository, which should be assigned to a `_var`.

void setPRNGSeed (const ssl::Current::PRNGseed& seed)

Sets a seed for the pseudo-random number generator used by the SSL layer.

Parameter	Description
seed	The `OctetSequence`seed for the PRNG.

void setPKprincipal (const CORBAsec::ASN1ObjectList chain,& const CORBAsec::ASN1Object& privkey,& const char* password);

This method is used in the client or the server to set the certificate chain and private key that must be used for the SSL connections. This is required for servers and optional for clients. Also look at the `peerAuthenticationMode` property documented in “Security Properties for C++.”

Parameter	Description
chain	The certificate chain.
privkey	The private key used for the SSL connection.
password	The password for the private key.

Exceptions

`CORBA::BAD_PARAM` if the user name or password is null.

void setCipherSuiteList (const ssl::CipherSuiteInfoList& list)

This method is used in the client or the server to specify the ciphers available for the SSL connections.

Parameter	Description
list	A comma-separated list of cipher suites.

ssl::CipherSuiteInfoList* listAvailableCipherSuites()

Returns a list of cipher suites available in VisiSecure. You are responsible for freeing memory.

Returns

A list of cipher suites.

ssl::CipherSuiteInfoList* getCipherSuiteList()

Returns the ciphers that are currently used by the SSL layer.

Returns

A list of cipher suites.

```
void setP12Identity (const CORBASEC::ASNIOBJECT& pks12cert, const char*
password)
```

Parameter	Description
pks12cert	PKCS#12 formatted data.
password	The private key password.

Certificate API

This API contains classes and methods for working with certificates.

class vbsec::CertificateFactory

This is a utility class for handling of certificates and keys.

Include File

The `vbssp.h` file should be included when you use this class.

Methods

CORBAsec::ASN1Object* importCRL(const CORBAsec::ASN1Object& b64crl) const

Import a crl in b64 encoded DER, so it is in the format ready for passing across method VBSSLContext::addTrustedCertificate() second argument

```
CORBAsec::X509CertList* importCertificateChain (const
CORBAsec::ASN1ObjectList& certs) const
```

Import the certification chain in the form of `CORBAsec::ASN1ObjectList` into `CORBAsec::X509CertList`, which could be use in `VBSSLContext`.

Parameter	Description
certs	`ASN1ObjectList` representation of the certificate chain.
b64crl	The CRL string in B64 text format to be converted

Returns

`CORBAsec::X509CertList` representation of the certificate chain for CORBA transportation.

CORBAsec::X509CertList* importCertificates (const CORBAsec::ASN1ObjectList& certs) const

Import the certification list in the form of `CORBAsec::ASN1ObjectList` into `CORBAsec::X509CertList`. Certificates need not be related to each other. The original order is preserved after importing.

Parameter	Description
certs	`ASN1ObjectList` representation of certificate list

Returns

`CORBAsec::X509CertList` representation of the certificate list.

CORBAsec::ASN1Object* importPrivateKey (const CORBAsec::ASN1Object& key) const

Convert the private key from BASE64 or PEM format to DER format.

Parameter	Description
key	`ASN1ObjectList` representation of private key object.

Returns

DER format of the private key.

CORBAsec::X509CertList* importCertificateChain (const CORBAsec::ASN1Object& pkcs12bytes, const CSI::UTF8String& password) const

Imports a certificate chain from pkcs12 binary.

Parameter	Description
pkcs12bytes	`ASN1ObjectList` representation of pkcs12 binary.
password	Password for the pkcs12 binary.

Returns

`CORBAsec::X509CertList` representation of the certificate chain.

CORBAsec::ASN1Object* importPrivateKey (const CORBAsec::ASN1Object& pkcs12bytes, const CSI::UTF8String& password) const

Import private key from pkcs12 binary.

Parameter	Description
pkcs12bytes	`ASN1ObjectList` representation of pkcs12 binary.
password	Password for the pkcs12 binary.

Returns

`CORBAsec::ASN1Object` representation of the private key object.

const CertificateFactory& printCertificate (const CORBAsec::X509Cert& certificate, std::ostream& stream) const

Print out the certification information into an output stream.

Parameter	Description
certificate	certificate to be printed.
stream	stream to which to output.

Returns

the `CertificateFactory`.

bool passwordForPrivatekey (const CSI::UTF8String& password, const CORBAsec::ASN1Object& privkey) const

Test if the given password can decrypt the given private key object.

Parameter	Description
password	The password to be tested.
privkey	The private key object to be decrypted.

Returns

`true` if decryption is successful, `false` if not.

class CORBAsec::X509Cert

This class represents an X509 certificate. When a client application binds to a CORBA object, the client uses this interface to obtain the server's certificate information. The server can use this interface to obtain the client's certification information, if the client has a certificate.

Include File

The `X509Cert_c.hh` file should be included when you use this class.

Methods

char* getSubjectDN()

Returns the subject DN contained in the certificate.

Returns

The subject name is returned in the following format:

```
CN=<value>, OU=<value>, O=<value>, L=<value>, S=<value>, C=<value>
```

char* getIssuerDN()

Returns the issuer DN contained in the certificate.

Returns

The subject name is returned in the following format:

```
CN=<value>, OU=<value>, O=<value>, L=<value>, S=<value>, C=<value>
```

CORBA::OctetSequence * getSignatureAlgorithm()

Returns the signature algorithm used in the certificate.

Returns

The signature algorithm used in the certificate.

CORBA::OctetSequence * getHash(CORBASEC::HashAlgorithm algorithm)

Returns a hash of the certificate.

Parameter	Description
algorithm	The hash algorithm. The possible values are: `CORBASec::MD5`, `CORBASec::MD2` and `CORBASec::SHA1`

Returns

A hash of the certificate using the specified algorithm.

CORBAsec::ASN1Object_ptr getDER()

Returns the DER encoded form of this certificate.

Returns

The ASN.1 DER encoded form of this certificate (assign to a `_var`).

CORBAsec::SerialNumberValue_ptr getSerialNumber()

Retrieves the serial number of the certificate.

Returns

The serial number of the certificate.

CORBAsec::X509CertExtensionList_ptr getExtensions()

Returns all the extensions available in this certificate as a list of X509CertExtension.

Returns

Returns all the extensions available in this certificate as a list of `X509CertExtension`. Or, if this certificate has no extensions, the method returns an array of length null. The extensions are not parsed.

CORBA::Boolean isValid (CORBA::ULong_out date)

Checks if a certificate date is between the valid start and end dates.

Parameter	Description
date	An out argument that is set to the expiration date of the certificate, using UNIX time format.

Returns

`true` if the certificate is valid, `false` otherwise.

CORBA::ULong startDate()

Gets the date from which a certificate's validity starts.

Returns

Returns an `int` representing the number of seconds from midnight, January 1st, 1970.

CORBA::ULong endDate()

Gets the expiration date of the certificate.

Returns

Returns an `int` representing the number of seconds from midnight, January 1st, 1970.

CORBA::Boolean equals (CORBAsec::X509Cert_ptr other)

Compares two CORBAsec::X509Cert certificates.

Parameter	Description
other	The other certificate to compare to this certificate.

Returns

Returns `true` (1UL) if the two certificates are identical; otherwise, returns `false` (0UL).

CORBA::Boolean isTrustpoint()

Checks if this certificate is a trustpoint—that is, if it is a trusted certificate

Returns

If the certificate is a trustpoint, returns `true`.

class CORBAsec::X509CertExtension

This class is an IDL structure that represents an X509 certificate extension, as follows:

```
struct X509CertExtension {
long seq;
sequence<long> oid;
boolean critical;
sequence<octet> value;
};
```

Parameter	Description
seq	A unique number of the extension in the certificate.
oid	The oid of the extension.
value	The value of the extension encoded according to the format specified by the oid.

Include File

The X509Cert_c.hh file should be included when you use this class.

QoP API

The following section details the Quality of Protection API provided with VisiSecure.

class vbsec::ServerConfigImpl

`ServerConfigImpl` is the implementation of the `csiv2::ServerQoPConfig`, which is an IDL structure as follows:

```
ServerConfigImpl (
    CORBA::Boolean disable,
    CORBA::Short transport,
```

```
    CORBA::Boolean trustInClient,
    csiv2::AccessPolicyManager* access_manager,
    const CORBA::StringSequence& realms = _available,
    CORBA::Short requiredIdentityType = csiv2::ServerQoPConfig::UP_OR_PK,
    CORBA::Boolean supportIdentityAssertion =
static_cast<CORBA::Boolean>(1)
);
```

Parameter	Description
disable	Whether or not to disable security.
transport	The transport mechanism to use. Valid values are: ■ `csiv2::CLEAR_ONLY`: no secure transport is necessary ■ `csiv2::SECURE_ONLY`: only secure connections are permitted ■ `csiv2::ALL`: any method of transport is allowed
trustInClient	Whether or not the target requests the client to authenticate. This value is set on CSIV2 layer.
access_manager	An access manager for the QoP implementation, an implementation of `csiv2::AccessPolicyManager` defined by the user. If null, it uses a default value.
realms	The available realms in which to implement the policy.
requiredIdentityType	The required identity for the QoP policy implementation. The default value is `csiv2::ServerQoPConfig::UP_OR_PK`. Possible values are: `csiv2:ServerQoPConfig::NO_ID`, `csiv2::ServerQoPConfig::UP`, `csiv2::ServerQoPConfig::PK`, `csiv2::ServerQoPConfig::UP_OR_PK` and `csiv2::ServerQoPConfig::UP_AND_PK`
supportIdentityAssertion	Whether or not the application supports Identity Assertion.

To define the `ServerQoPPolicy`, you create this object which defines the various characteristics of the policy.

Include File

The `CSIV2Policies.h` file should be included when you use this class.

class ServerQoPPolicyImpl

`ServerQoPPolicyImpl` is the implementation of the `csiv2::ServerQoPPolicy`. The `ServerQoPPolicyImpl` object impacts the QoP behaviour of the server.

Include File

The `CSIV2Policies.h` file should be included when you use this class.

Methods

ServerQoPPolicyImpl (const csiv2::ServerQoPConfig_var& conf);

Constructor of the `ServerQoPPolicyImpl` object.

Parameter	Description
conf	`ServerQoPConfig` object which contains the designed QoP configuration.

virtual csiv2::ServerQoPConfig_ptr config();

Get the ServerQoPConfigImpl object from the ServerQoPPolicyImpl.

Returns

The `ServerQoPConfigImpl` object from the `ServerQoPPolicyImpl`.

class vbsec::ClientConfigImpl

`ClientConfigImpl` is the implementation of the `csiv2::ClientQoPConfig`. To define the `ClientQoPPolicy`, you create this object which defines the various characteristics of the policy.

Include File

The `CSIV2Policies.h` file should be included when you use this class

Methods

ClientConfigImpl (const CORBA::Short transport, const CORBA::Boolean trustInTarget)

Constructor of ClientConfigImpl object.

Parameter	Description
transport	The transport mechanism to use. Valid values are: ■ `csiv2::CLEAR_ONLY`: no secure transport is necessary ■ `csiv2::SECURE_ONLY`: only secure connections are permitted ■ `csiv2::ALL`: any method of transport is allowed
trustInTarget	Whether or not to require the client to authenticate.

class vbsec::ClientQoPPolicyImpl

`ClientQoPPolicyImpl` is the implementation of the `csiv2::ClientQoPPolicy`. The `ClientQoPPolicyImpl` object impacts the QoP behaviour of the server.

Include File

The CSIV2Policies.h file should be included when you use this class.

Methods

ClientQoPPolicyImpl(const csiv2::ClientQoPConfig_var& conf);

Constructor for `ClientQoPPolicyImpl` object.

Parameter	Description
conf	`ClientConfigImpl` object to be use for the policy.

virtual csiv2::ClientQoPConfig_ptr config();

Returns

The `ClientConfigImpl` object of this `ClientQopPolicyImpl`.

Authorization API

The following section describes the classes and methods used for authorization in VisiSecure.

class csiv2::AccessPolicyManager

`AccessPolicyManager` is used define your Access Policy for authorization a client's method calls.

Include File

The `CSIV2Policies.h` file should be included when you use this class.

Methods

char* domain()

Returns the authorization domain name for the `AccessPolicyManager`.

Returns

The authorization domain name for the object that uses this `AccessPolicyManager`.

csiv2::ObjectAccessPolicy* getAccessPolicy (PortableServer_ServantBase* servant, const PortableServer::ObjectId& id, const CORBA::OctetSequence& adapter_id)

Returns the objectAccessPolicy for the servant with the `objectId` (id) and poa id.

Parameter	Description
servant	The CORBA servant object.
id	the id of the servant object.
adapter_id	The poa id of the servant object.

Returns

`ObjectAccessPolicy` of the servant object.

class csiv2::ObjectAccessPolicy

This class represents the access policy from `AccessPolicyManager`.

Include File

The `CSIV2Policies.h` file should be included when you use this class.

Methods

CORBA::StringSequence* getRequiredRoles (const char* method)

Returned the list of required roles to access the method.

Parameter	Description
method	The method name of interest.

Returns

A list of required roles to access the method.

char* getRunAsRole (const char* method)

Return the run-as role for the method. This method is not used in this release.

Parameter	Description
method	The method name of interest.

Returns

The run-as role configured to access the method.

42

Security SPI for C++

This section describes the Service Provider Interface (SPI) classes as defined for the VisiSecure for C++. These SPI classes provide advanced security functionality and allow other security providers to plug their own implementation of security services into VisiSecure for their use.

Plugin Mechanism and SPIs

VisiSecure for C++ provides interfaces for you to plug in your own security implementations. In order for the ORB to find your implementation, all plug-ins must use the `REGISTER_CLASS` macro provided by the VisiSecure to register your classes. The name of the class must be specified in full, together with its namespace upon registration. Namespace must be specified in a normalized form supported by VisiSecure, using either a '.' or '::' separated-string starting from the outer namespace. For example:

```
MyNameSpace {
  class MyLoginModule {
    ......
  }
}
```

Thus `MyLoginModule` shall be specified as either `MyNameSpace.MyLoginModule` or `MyNameSpace::MyLoginModule`.

There are six pluggable components:

- **LoginModules:** You can implement your own login models by extending the `vbsec::LoginModule`. To use the login module, you need to set it in the authentication configuration file, just like any other login module.
  Additionally, you will also need to register your class using `MAP_LOGIN_MODULE (<CLASS_NAME>)` macro in your implementation file which would enable security runtime to dynamically load your login module.
- **Callback handlers:** You can implement your own callback by extending the `vbsec::CallbackHandler`. To use the callback, you need to set it in the authentication configuration file, just like any other callback handler.
  Additionally, you will also need to register your class using `MAP_CALLBACK_HANDLER(<CLASS_NAME>)` macro in your implementation file which would enable security runtime to dynamically load your callback handler.

- **Identity adapters, Mechanism adapters, and Authentication Mechanisms:** these interfaces are provided for users to implement their own authentication mechanisms and identity interpretations. `IdentityAdaptor` is to interpret identities, `MechanismAdaptor` is a specialized identity adaptor which also changes target information. `AuthenticationMechanism` is a pluggable service to authenticate users.

  To use these plug-ins, you need to set the `vbroker.security.identity.xxx` properties to define the plug-ins and their properties. For example, an identity adapter or mechanism adapter must specify:

  ```
  vbroker.security.identity.adapters=MyAdapter
  vbroker.security.adapter.MyAdapter.property1=value1
  vbroker.security.adapter.MyAdapter.property2=value1
  ```

  while an authentication mechanism must provide:

  ```
  vbroker.security.identity.mechanisms=MyMechanism
  vbroker.security.adapter.MyMechanism.property1=value1
  vbroker.security.adapter.MyMechanism.property2=value2
  ```

  The properties specified must be passed to the user plug-in during initialization as a string map. The map contains truncated key/value pair like `property1, value1`.

- **Attribute codec:** This allows you to plug in an attribute codec to encode and decode attributes in their own format. VisiSecure for C++ has one build-in codec, the ATS codec.

  To use your codec plug-in, you need to set properties to define the codecs and their properties. For example:

  ```
  vbroker.security.identity.attributeCodecs=MyCodec
  vbroker.security.adapter.attributeCodec.property1=xxx
  vbroker.security.adapter.attributeCodec.property2=xxx
  ```

  The properties specified will be passed to the user plug-in during initialization as a string map.

- **Authorization service provider:** You can plug-in an authorization service for each authorization domain. VisiSecure has its default implementation, which uses the rolemap. Like the other pluggable services, you will need to define the authorization service with properties which are then passed as string maps. For example:

  ```
  vbroker.security.auth.domains=MyDomain
  vbroker.security.domain.MyDomain.provider=MyProvider
  vbroker.security.domain.MyDomain.property1=xxx
  vbroker.security.domain.MyDomain.property2=xxx
  ```

- **Trust provider:** This allows you to plug in an assertion trust mechanism. Assertion can happen in multi-hop scenario, or explicitly called through assertion API. The server can have rules to determine whether the peer is trusted to make the assertion or not. The default implementation uses the property setting to configure trusted peers on the server side. During the runtime, the peer must pass authentication and authorization in order to be trusted to make assertions.

  Like the other pluggable services, you will need to define the authorization service with properties which are then passed as string maps. For example:

  ```
  vbroker.security.trust.trustProvider=MyProvider
  vbroker.security.trust.trustProvider.MyProvider.property1=xxx
  vbroker.security.trust.trustProvider.MyProvider.property2=xxx
  ```

  There can be only one trust provider specified for the whole security service.

Providers

Each provider instance is created by the VisiSecure using a Java reflection API. After the instance has been constructed, the `initialize` method, which must be provided by the implementer, is called passing in a map of options specific for the implementation. The options entries are defined by the implementers of the particular provider. Users specify the options in a property file and the VisiSecure parses the property and passes the options to the corresponding provider. The following table shows the properties for plugging in different provider implementations.

Module Name	Property to set	Interface to implement	Options Prefix
IdentityAdapter	`vbroker.security.identity.adapters`	`vbsec::IdentityAdapter`	`vbroker.security.identity.adapter.<name>`
AuthenticationMechanism	`vbroker.security.identity.mechanisms`	`vbsec::AuthenticationMechanisms`	`vbroker.security.identity.mechanism.<name>`
AttributeCodec	`vbroker.security.identity.attributeCodecs`	`vbsec::AttributeCodec`	`vbroker.security.identity.attributeCodec.<name>`
TrustProvider	`vbroker.security.trustProvider`	`vbsec::TrustProvider`	`vbroker.security.trust.trustProvider.<name>`

In the preceding table:

- The first column lists the provider module names.
- The second column lists the property you set to define each module. Use a comma to separate multiple modules. For example, the following property has two additional IdentityAdapter implementations installed for the ORB:

  ```
  vbroker.security.identity.adapters=ID_ADA1,ID_ADA2
  ```

- The third column gives the interface each implementation must implement. The interface defines a contract between the implementers and the core VisiSecure.
- The final column gives the options prefix for the specific module. The ORB parses the property file and passes the corresponding entries to each of the modules in the initial method as the (Map options) parameter.
- For example, for the `ID_ADA1 IdentityAdapter` defined in the previous example, all the entries with the `vbroker.security.identity.adapters.ID_ADA1` prefix will be passed to the initial method of `ID_ADA1` IdentityAdapter.

Providers and exceptions

During the initialization, if anything goes wrong, the `initialize` method should throw an instance of `InitializationException`. For certain categories of providers, there can be multiple instances with different implementations co-existing. Each of them is identified by the name within the VisiSecure system, which is passed as the first parameter in the `initialize` method. While for some categories of providers there can be only one instance existing for the whole ORB (such as in the case of the TrustProvider, in this case, the `initialize` method has only one single parameter -the options map.

vbsec::LoginModule

The LoginModule serves as the parent of all login modules. The user plug-in login modules must extend this class. The login modules are configured in the authentication configuration file and are called during the login process. The login modules are responsible of authenticating the given subject and associating the relevant Principals and Credentials with the subject. It is also responsible for removing and disposing of such security information during the logout.

Include File

The vbauthn.h file should be included when you use this class.

Methods

```
void initialize (Subject* subj=0,
                 CallbackHandler *handler=0,
                 LoginModule::states* sharedStates=0,
                 LoginModule::options* options=0)
```

This method initializes the login module.

Arguments

This method utilizes the following four arguments:

- subj: the subject to be authenticated.
- handler: the callback handler to use.
- sharedStates: the additional authentication state provided by other login modules. Currently not used.
- options: configuration options specified in the authentication configuration file.

Returns

Void.

```
bool login()
```

Performs the login. This is called during the login process. The login module shall authenticate the subject located in the module and determine if the login is successful.

Returns

true if the login succeeds, false otherwise.

```
bool logout()
```

Performs the logout. This is called during the logout process. The login module shall logout the subject located in the module and determine if the logout is successful. The login module may remove any credentials or identities that were established during the login and dispose them.

Returns

true if the logout succeeds, false otherwise.

```
bool commit()
```

Commits the login. This is part of the login process, called when the login succeeds according to the configuration options specified in the pertinent login modules. The login module then associates relevant Principals and Credentials with the Subject located in the module if its own authentication attempt succeeded. Or if not, it shall remove and destroy any state that was saved before.

Returns

true if the commit succeeds, false otherwise.

```
bool abort()
```

Aborts the login. This is part of the login process, called when the overall login fails according to the configuration options specified in the login modules. The login module shall remove and destroy any state that was saved before.

Returns

`true` if the abort succeeds, `false` otherwise.

vbsec::CallbackHandler

`The CallbackHandler` is the mechanism that produces any necessary user callbacks for authentication credentials and other information. Seven types of callbacks are provided. There is a default handler that handles all callbacks in an interactive text mode.

Include file

The `vbauthn.h` file should be included when you use this class.

Methods

```
void handle (Callback::array& callbacks)
```

Handle the callbacks.

Arguments

The array of `callbacks` to be processed.

Returns

Void.

vbsec::IdentityAdapter

IdentityAdapter binds to a particular mechanism. The main purpose of an IdentityAdapter is to interpret identities that are specific to a mechanism. It is used to perform the decoding and encoding between the mechanism-specific and the mechanism-independent representations of the entities.

IdentityAdapters included with the VisiSecure

The following IdentityAdapters are provided with the VisiSecure:

- `AnonymousAdapter`, with the name "`anonymous`"
- `DNAdapter`, with the name "`DN`"
- `X509CertificateAdapter` (as an implementation of the sub-interface AuthenticationMechanism)
- `GSSUPAuthenticationMechanism` (as an implementation of the sub-interface AuthenticationMechanism)

Methods

```
Virtual void initialize (const std::string& name, ::vbsec::InitOptions&) =0;
```

This method initializes the IdentityAdapter with the given name and the set of options.

Arguments

This method takes the following two arguments:

- The IdentityAdapter `name`.
- A set of `InitOptions` for the specified IdentityAdapter.

Exceptions

Throws InitializationException if initialization fails.

```
virtual std::string getName() const=0;
```

This returns the name of the IdentityAdapter.

Returns

The name of the IdentityAdapter.

Exceptions

none

```
virtual ::CSI::IdentityToken* exportIdentity(::vbsec::Subject&, ::CSI::IdentityToken&)
=0;
```

Exports the identity of the IdentityAdapter as an IdentityToken.

Arguments

The subject whose identity is to be exported.

Returns

An IdentityToken data.

Exceptions

Throws NoCredentialsException if no credentials recognized by this IdentityAdapter are found in the subject.

```
virtual void importIdentity (::vbsec::Subject&, ::CSI::IdentityToken&) =0;
```

Imports the IdentityToken and populates the caller subject with the appropriate principals associated with this identity.

Arguments

The subject whose identity is to be imported.

Exceptions

Throws NoCredentialsException if no credentials recognized by this IdentityAdapter are found in the subject.

```
virtual ::vbsec::Privileges* getPrincipal (::vbsec::Subject&anp;) =0;
```

Returns a Principal representing this identity. This method is used for interfacing with EJBs and servlets.

Arguments

The principal subject.

Returns

A principle object.

Exceptions

none

```
virtual ::vbsec::Privileges* getPrivileges (::vbsec::Subect&) =0;
```

Arguments

The target subject.

Returns

The privilege attributes for this target subject recognized by this IdentityAdapter.

Exceptions

none

```
virtual ::vbsec::setPrivileges (::vbsec::Privileges*) =0;
```

This methods sets the privilege attribute for the identity.

Arguments

The privilege attribute to be set for the identity.

Exceptions

none

```
virtual void deleteIdentity (::vbsec::Subject&) =0;
```

This method deletes the principals and the credentials associated with the specified target subject.

Arguments

The target subject for which the principals and the credentials recognized by this IdentityAdapter are to be deleted.

Exceptions

none

vbsec::MechanismAdapter

Extending from `IdentityAdapter`, a `MechanismAdapter` has the additional capability of changing the target information. This is very useful in the case where the mechanism used in a remote site is not available locally. Therefore, the local identity must be adapted before sending to the remote site.

In the out-of-box installation of VisiSecure, there is no class direct implementation of `MechanismAdapter`, while a few classes implement the sub-interface AuthenticationMechanism, which in turn gives the support of this interface.

Methods

```
virtual const ::CSI::StringOID_var getOid() const =0;
```

Returns a string representation of the mechanism OID. For example, the string representation for a GSSUP mechanism would be `oid:2.23.130.1.1.1`.

Returns

The mechanism OID string.

Exceptions

none

```
virtual ::vbsec::Target* getTarget (const std::string& realm, const
std::vector<AppConfigurationEntry*>&) =0;
```

Given a realm name and a list of AppConfigurationEntry objects, returns the corresponding target.

Arguments

This method takes the following two arguments:

- A realm name.
- A list of `AppConfigurationEntry` objects.

Returns

Returns the corresponding target object.

Exceptions

none

```
virtual ::vbsec::Target* getTarget (const ::CSI::GSS_NT_ExportedName&) =0;
```

Returns a Target object representing the encoded target representation.

Arguments

A Target encoded in GSS Mechanism-Independent Exported Name format (as defined in [IETF RFC2743]).

Returns

A Target object.

Exceptions

none

vbsec::AuthenticationMechanisms

This class represents a full-fledged mechanism which provides all the functionality needed to support an authentication mechanism in conjunction with the CSIv2 protocol.

Included with VisiSecure are the following implementations for GSSUP based and X509 Certificate based authentication mechanisms respectively:

- GSSUPAuthenticationMechanism
- X509CertificateAdapter

In addition to the methods inherited from its super interfaces, `AuthenticationMechanism` also has the following categories of methods defined.

Credential-related methods

Use these methods to acquire and/or destroy credentials.

```
virtual ::vbsec::Subject* acquireCredentials (::vbsec::Target&,
::vbsec::CallbackHandler*) =0;
```

This method acquires credentials for a given target. The credentials acquired depend on the mechanism and the information it requires for authentication.

Arguments

This method takes the following two arguments:

- A Target object.
- The callback handler to be used to communicate with the user for acquiring the credentials for this Target.

Returns

The Subject containing the acquired credentials (will be null in the case where the operation fails).

Exceptions

none

```
virtual ::vbsec::Subject* acquireCredentials (const std::string& target,
::vbsec::CallbackHandler*) =0;
```

This method acquires credentials for a given string representation of the Target. The credentials acquired depend on the mechanism and the information it requires for authentication.

Arguments

This method takes the following two arguments:

- A string representation of the Target.
- The corresponding callback handlers used to communicate with user for acquiring the credential.

Returns

A subject object containing the acquired credentials (It will be null in cases where the operations fail).

Exceptions

none

```
virtual void destroyPrivateCredentials (::vbsec::Subject&) =0;
```

This method destroys the private credentials of the specified subject.

Arguments

The subject for which the private credentials are to be destroyed.

Exceptions

none

Context-related methods

```
virtual ::CORBA::OctetSeq* createInitContext (::vbsec::Subject&) =0;
```

Returns a mechanism-specific client authentication token. The token represents the authentication credentials for the specified target.

Arguments

The target subject.

Returns

The authentication token for the specified target subject.

Exceptions

Throws NoCredentialsException if no authentication credentials recognized by this mechanism exist in this Subject.

```
virtual ::vbsec::Target* processInitContext (::vbsec::Subject&, ::CORBA::OctetSeq&) =0;
```

This method consumes the mechanism-specific client authentication token. The initial authentication token is decoded and the method populates the given subject with the corresponding authentication credentials.

Arguments

The subject to be populated with authentication credentials.

Exceptions

none

```
virtual ::CSI::GSSToken* createFinalContext (::vbsec::Subject&) =0;
```

This method creates a final context token to return to a client.

Arguments

The Subject.

Returns

A final context token.

Exceptions

none

```
virtual void processFinalContext (::vbsec::Subject&, ::CORBA::OctetSeq&) =0;
```

Consumes a final context token returned by the server.

Arguments

The target subject.

Exceptions

none

```
virtual ::CSI::GSSToken* createErrorContext (::vbsec::Subject&) =0;
```

Creates an error context token in the case of an authentication failure.

Arguments

The target subject.

Returns

An error context token.

Exceptions

none

```
virtual ::vbsec::Subject* processErrorContext (::vbsec::Subject&, ::CSI::GSSToken&,
::vbsec::CallbackHandler*) =0;
```

Consumes an error token returned from server. The callback handler is used to interact with a user trying to re-acquire credentials. If credentials are required, the client-side security service attempts to establish the context again.

Arguments

This method takes the following two arguments:

- A target subject.
- A callback handler.

Exceptions

none

vbsec::Target

This class gives the runtime representation of a target authenticating principal. The context includes names for the target required in different scenarios, such as the display name, or the DER representation of the OID.

Methods

```
virtual std::string getName () const =0;
```

This method returns the display name of the target.

Returns

The target name string.

Exceptions

none

```
virtual ::CSI::OID getOid () const =0;
```

This method returns the target OID.

Returns

The target OID string.

Exceptions

none

```
virtual ::CORBA::OctetSeq getEncodedName () const =0;
```

This method returns the mechanism-specific encoding of the target name.

Returns

The encoded target name.

Exceptions

none

vbsec::AuthorizationServicesProvider

The implementer of the Authorization Service provides the collection of permission objects granted access to certain resources. Whenever an access decision is going to be made, the AuthorizationServicesProvider is consulted. The Authorization Service is closely associated with the Authorization domain concept. An Authorization Service is installed for each Authorization domain implementation, and functions only for that particular Authorization domain.

The `AuthorizationServicesProvider` is initialized during the construction of its corresponding Authorization domain. Use the following property to set the implementing class for the AuthorizationServicesProvider:

```
vbroker.security.domain.<domain-name>.provider
```

During the runtime, this property is loaded by the way of Java reflection.

Another import functionality of the Authorization Service is to return the run-as alias for a particular role given. The security service is configured with a set of identities, identified by aliases. When resources request to “run-as” a given role the AuthorizationService again is consulted to return the alias that must be used to “run-as” in the context of the rules specified for this authorization domain.

Methods

```
virtual void initialize (const std::string& name, ::vbsec::InitOptions& options) =0;
```

This method initializes an Authorization Services provider.

Arguments

This method takes the following arguments:

- A provider name.
- The provider options.

In addition to the provider's options, the following information is passed to facilitate the interaction between this Authorization Service provider and the VisiBroker ORB:

Name	Description
ORB	The ORB instance used for the current system.
Logger	A SimpleLogger instance used for login in the current system.
LogLevel	An integer value denoting the security logging level.

Exceptions

Throws InitializationException if initialization of the Authorization provider fails.

```
virtual std::string getName() const =0;
```

Returns the name for this Authorization Service implementation.

Returns

The Authorization Service name.

Exceptions

none

```
virtual ::vbsec::PermissionCollection* getPermissions (const ::vbsec::Resource*
resource, const ::vbsec::Privileges* callerPrivileges) =0;
```

Returns a homogeneous collection of permission attributes for the given privileges as well as the resource upon which the access is attempted.

Arguments

This method takes the following two arguments:

- The caller Privileges.
- The resource object upon which access is to be attempted.

Returns

A PermissionCollection object represents this subject's Permissions.

Exceptions

none

vbsec::Resource

The Resource interface gives a generic abstraction of resource. The resource can be anything upon which the access will be made, such as a remote method of a CORBA object, or a servlet which is essentially a resource.

Methods

```
virtual std::string getName () const =0;
```

Returns the string representation of the resource being accessed.

Returns

Name of the resource.

Exceptions

none

vbsec::Privileges

The Privileges class gives an abstraction of the privileges for a subject. It is the container of authorization privilege attributes, such as Distinguished Name (DN) attributes, and such. The AuthorizationService makes the decision on whether the subject has permission to access the certain resource based on the privileges object of the subject.

The privileges object is stored inside the subject as one of the PublicCredentials. At the same time, privileges hold one reference to the referring subject. Privileges also contain a DN attributes map, as well as a map of other authorization attributes.

The Privileges class implements the `javax.security.auth.Destroyable` interface.

Constructors

```
Privileges (const std::string& name, ::vbsec::Subject& subject);
```

This constructor creates a privileges object with the given name and associates it with the given subject.

Arguments

The method takes the following two arguments:

- Name of the Privileges object, which is actually the associated Subject's name.
- The target subject.

Exceptions

none

Methods

```
::vbsec::Subject& getSubject() const ;
```

This method returns the subject that the privileges object represents.

Returns

The target subject.

Exceptions

none

```
std::string getSubjectName() const;
```

This method returns the name of the associated subject object.

Returns

The target subject.

Exceptions

none

```
const ::vbsec::ATTRIBUTE_MAP& getAttributes() const ;
```

This method returns the attribute map of the user.

Returns

The user's attribute map.

Exceptions

none

```
void setDBAttributes (const ::vbsec::ATTRIBUTE_MAP& map);
```

This method updates the DN Attributes of the user.

Arguments

The new DN Attributes Map.

Note

After the DN Attributes Map has been set, the Privileges object will set the underlying DN Attributes Map as unmodifiable.

Exceptions

none

```
const ::vbsec::ATTRIBUTE_MAP* getDNAttributes() const;
```

This method returns the DN Attributes of the Privileges object, which can be null.

Returns

User's DN Attributes map, which is not modifiable.

Exceptions

none

```
bool isDestroyed() const;
```

This method checks whether the privileges object has been destroyed or not.

Returns

true|false

Exceptions

none

```
std::string toString() const;
```

This method overrides the default `toString` implementation of `java.lang.Object`, and returns “Privileges for <subject name>” information.

Returns

List of privileges for each subject name.

Exceptions

none

vbsec::AttributeCodec

The AttributeCodec objects are responsible for encoding and decoding privileges attributes of a given subject. This allows clients and servers to communicate the

privilege information to each other. Though the privilege information is used as the basis for the Authorization decision-making process, AttributeCodec selection is based on the information presented in the IOR published by the server. Inside the IOR, the server publishes information on the encoding scheme supported, while clients select an AttributeCodec that supports the given encoding.

All the AttributeCodecs implementations are registered with the IdentityServices, which are called upon during the import/export of the authorization elements process.

Methods

```
virtual void initialize (const std::string& name, vbsec::InitOptions& options) =0;
```

This method initializes this instance of the AttributeCodec implementation. There can be multiple implementations existing in one ORB, and each is addressed internally using the name provided.

Arguments

This method takes the following arguments:

- A string of AttributeCodec implementation names.
- Provider options.

For the provider's options, the following additional information is also passed during the initialization:

Name	Description
ORB	The ORB instance used for the current system.
Logger	A SimpleLogger instance used for the current system for the purpose of logging.
LogLevel	An integer value denoting the security logging level.

Exceptions

Throws InitializationException if initialization of this AttributeCodec object fails.

```
virtual std::string getName() const =0;
```

This method returns the name of the provider implementation.

Returns

The provider name string.

Exceptions

none

```
virtual CSIIOP::ServiceConfigurationList* getPrivilegeAuthorities() const =0;
```

This method returns a list of supported privilege authorities.

Returns

A list of privilege authorities.

Exceptions

none

```
4. virtual CSI::AuthorizationElementType getSupportedEncoding() const = 0;
```

This method returns the supported AuthorizationElement type.

Returns

An AuthorizationElement type.

Exceptions

none

```
virtual bool supportsClientDelegation() const =0;
```

Returns whether this implementation supports ClientDelegation.

Returns

true|false

Exceptions

none

```
virtual CSI::AuthorizationToken* encode (const CSIIOP::ServiceConfigurationList&
privilege_authorities, vbsec::Privileges& caller_privileges, vbsec::Privileges&
asserter_privileges) =0;
```

This method encodes privileges as AuthorizationElements. This method encodes the privilege attributes of the given caller and the given asserter, if there is one. It will extract the privilege information from the subject and privilege map of the caller and the asserter.

Additionally, an implementation of the AttributeCodec (if supports ClientDelegation) may choose to verify whether the asserter is allowed to assert the caller based on the client delegation information presented by this caller.

Arguments

This method takes the following arguments:

- A set of caller privileges attributes.
- A set of asserter privileges attributes.

Returns

Encoded caller and asserter privileges.

Exceptions

Throws NoDelegationPermissionException if the assertion is not allowed.

```
virtual void decode (const ::CSI::AuthorizationToken& encoded_attributes,
vbsec::Privileges& caller_privileges, vbsec::Privileges& asserter_privileges) =0;
```

This method decodes authorization elements and populates the corresponding privileges objects. This is the inversion process of the encode method. When a server receives a set of encoded AuthorizationElements, it passes these elements to the AttributeCodec for interpretation. Based on the encoding method, one particular AttributeCodec consumes the attributes it understands. It may update the caller's or asserter's Privileges, or may add RolePermission directly to the subject's public credentials.

Arguments

This method takes the following arguments:

- A set of encoded Authorization Elements.
- A set of caller privileges.
- A set of asserter privileges.

Returns

This method returns nothing. Upon a successful processing, this AttributeCode object updates the caller's or asserter's Privileges maps as appropriate based on the information available in the authorization elements.

Exceptions

Throws NoDelegationPermissionException if the assertion is not authorized.

vbsec::Permission

`Permission` represents the authorization information to access resources. Every permission has a name, which can be interpreted only by the actual implementation.

Include file

The `vbsecspishared.h` file should be included when you use this class.

Methods

```
bool implies (const Permission& p) const
```

Evaluate if the permission implies another given permission. This is used during the authorization process to determine if the caller permissions imply the permissions required by the resource. Access will be granted if the caller permissions imply the required permission, or denied if not.

Arguments

The permission `p` to be evaluated.

Returns

`true` if the permission implies an existing permission, `false` otherwise.

```
bool operator==(const Permission& p) const
```

Checks if the permission equals another given permission.

Arguments

The permission `p` to be evaluated.

Returns

`true` if the permissions are equal, `false` otherwise.

```
std::string getName () const
```

Gets the name of the permission.

Returns

The name of the permission.

```
std::string getActions () const
```

Get the actions of the permission as a string. It is only interpreted by the actual implementation.

Returns

The string representation of the action for the permission.

```
std::string toString () const
```

Get the string representation of the permission.

Returns

The string representation of the permission.

vbsec::PermissionCollection

`PermissionCollection` represents a collection of permissions.

Include file

The vbsecspishared.h file should be included when you use this class.

Methods

```
bool implies (const Permission& p) const
```

Evaluate if the PermissionCollection implies the given permission.

Arguments

the permission p to be evaluated.

Returns

true if the PermissionCollection implies the given one, false otherwise.

vbsec::RolePermission

The RolePermission class provides the basis for authorization and trust in the VisiSecure system.

Constructors

```
RolePermission (const std::string& role)
```

This constructor creates a RolePermission object representing a logic role.

Arguments

A logical role string this RolePermission object represents.

Returns

A RolePermission object.

Exceptions

none

Methods

```
virtual bool implies (const Permission& permission) const;
```

This method checks whether the permission object passed in implies this RolePermission object. The check is based on strict equality, as the method only returns true (implies) when ALL the following conditions exist:

1. the permission object given is an instance of RolePermission, and
2. the name of the permission object given equals the name of this RolePermission.

Arguments

A Permission object to check.

Returns

True|False

Exceptions

none

```
virtual std::string getActions() const;
```

This method returns the action associated with this RolePermission.

Returns

Always returns null, since there are no actions associated with a RolePermission object.

Exceptions

none

vbsec::TrustProvider

When a remote peer (server or process) makes identity assertions in order to act on behalf of the callers, the end-tier server needs to trust the peer to make such assertions. This is meant to prevent untrusted clients from making assertions.

The key method is `isAssertionTrusted`, which is called to determine whether the assertion is trusted given the caller subject and asserter's privileges. This method is called (by the underline implementation) after the corresponding authorization elements transmitted from a client to the server have been consumed.

You use the TrustProvider class to implement trust rules which determine whether the end-tier server accepts identity assertions from a given asserting subject. The TrustProvider class is very closely related to the implementation of the AttributeCodec objects and the privileges. For example, it is possible to provide the decision-making implementation as follows:

1. Provide class implementations representing a proxy endorsement attribute,
2. AttributeCodec implements the necessary logic then passes the attributes and imports them to the caller subject on the server-side. It is also necessary to return `true` for the method supportsClientDelegation defined in the AttributeCodec interface.
3. Provide the method implementation based on the proxy endorsement attribute of the caller and the privileges of the asserter.

This type of evaluation of trust, which is based on rules provided by the caller, is referred to as Forward Trust. Backward Trust is when the evaluation of trust is based on the rules of the target. Backward Trust is the default provided with the VisiSecure installation. For more information, see “Trust assertions and plug-ins”.

Methods

```
virtual void initialize (::vbsec::InitOptions&, std::map<std::string, std::string>&) =0;
```

This method initializes the `TrustProvider`. There can be only one instance of the TrustProvider implementation existing for each process.

Arguments

For the provider's options, the following additional information is also passed during the initialization:

Name	Description
`ORB`	The ORB instance used for the current system.
`Logger`	A `SimpleLogger` instance used for the current system for the purpose of logging.
`LogLevel`	An integer value denoting the security logging level.

Exceptions

Throws InitializationException if initialization of the TrustProvider fails.

```
virtual bool isAssertionTrusted (const ::vbsec::Subject&, const ::vbsec::Privileges&)
=0;
```

This method verifies whether an assertion of the caller by the asserter with the provided privileges is trusted or not. The implementation makes use of the internal trust rules for this process to determine the validity of the assertion.

Arguments

This method takes the following two arguments:

- The caller.
- The set of asserter privileges.

Returns

true|false

Exceptions

none

vbsec::InitOptions

InitOptions is a data structure passed to user plug-ins during initialization calls that facilitates the initialization process.

Include file

The vbsecspishared.h file should be included when you use this class.

Data Members

```
std::map<std::string, std::string>* options
```

A string map containing name/value pair presenting parsed property setting.

```
::PortableInterceptor::ORBInitInfo* initInfo
```

Object representing the ORB initialization information.

```
::IOP::Codec* codec
```

An IOP Codec object.

```
::vbsec::SimpleLogger* logger
```

A logger object.

```
int logLevel
```

The log level currently configured for the security service.

vbsec::SimpleLogger

SimpleLogger is a mechanism to log information of various levels. Currently, it supports four different levels: LEVEL_WARNING, LEVEL_NOTICE, LEVEL_INFO, and LEVEL_DEBUG, with increasing detailed information. There is only one logger in the whole security service.

Include file

The vbsecspishared.h file should be included when you use this class.

Methods

```
::std::ostream& WARNING()
```

Returns the logging output stream for warning messages.

Returns

The logging output stream for `LEVEL_WARNING`.

```
::std::ostream& NOTICE()
```

Returns the logging output stream for notice messages.

Returns

The logging output stream for `LEVEL_NOTICE`, or a fake stream if the log level is set below `LEVEL_NOTICE`.

```
::std::ostream& INFO()

Returns the logging output stream for info messages.
```

Returns

The logging output stream for `LEVEL_INFO`, or a fake stream if the log level is set below `LEVEL_INFO`.

```
::std::ostream& DEBUG()
```

Returns the logging output stream for debug messages.

Returns

The logging output stream for `LEVEL_DEBUG`, or a fake stream if the log level is set below `LEVEL_DEBUG`.

A

VisiSecure Error Codes

This appendix provides information about error codes for VisiSecure.

The tables in the subsequent sections lists most of the minor codes and their corresponding descriptions that accompany the CORBA system exceptions thrown from within VisiSecure Java and/or C++ module. ERROR_ID helps you to identify/ represent these errors inside the code as illustrated in the following sections:

Modifying Minor Codes in C++:

The header file "`vbsecminors.h`" needs to be included for ERROR_IDs to be made available. Then, in the code, you can use `vbsec::MinorCodes::<ERROR_ID>\`

For example;

`vbsec::MinorCodes::ERROR_PARSING_CERTIFICATE` helps to identify a given error (returned in the form of minor code as part of a CORBA system exception).

Modifying Minor Codes in Java:

You can use `com.borland.security.util.MinorCodes.<ERROR_ID>`

For example: `com.borland.security.util.MinorCodes.ERROR_PARSING_CERTIFICATE` can be used to identify a given error (returned in the form of minor code as part of a CORBA system exception).

The static method `String getMinorCodeDescription` (int minor) of class `com.borland.security.util.MinorCodesUtil` can be used to fetch the brief textual description of the error code.

General Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422100	INTERNAL_ERROR	Unrecoverable internal runtime error	NO_PERMISSION BAD_PARAM INTERNAL
0x56422101	UNTRUSTED_ASSERTION	Peer has insufficient privilege to assert the caller. See the description of the following properties/ scheme: `vbroker.security.trust.*`	NO_PERMISSION

PKI Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422201	ERROR_PARSING_CERTIFICATE	Unknown error when parsing the certificate data Example cause: corrupted DER/Base64 data often happens when copying over via ftp that may involve translation for UNIX<->DOS CR-LF	NO_PERMISSION BAD_PARAM
0x56422202	ERROR_PARSING_PRIVATE_KEY	Unknown/unclassified error when parsing the private key data. Example cause: corrupted DER/Base64 data often happens when copying over via ftp that may involve translation for UNIX<->DOS CR-LF	BAD_PARAM

SSL Errors

The followings are the translation of SSL Error alerts as per the SSL (see http://wp.netscape.com/eng/ssl3/ssl-toc.html)

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422221	UNKNOWN_CERTIFICATE	Translated from SSL alerts: certificate_unknown	NO_PERMISSION BAD_PARAM
0x56422222	UNSUPPORTED_CERTIFICATE	Translated from SSL alerts: unsupported_certificate	NO_PERMISSION
0x56422223	BAD_CERTIFICATE	Translated from SSL alerts: bad_certificate	NO_PERMISSION BAD_PARAM
0x56422224	CERTIFICATE_REVOKED	Translated from SSL alerts: certificate_revoked	NO_PERMISSION
0x56422225	CERTIFICATE_EXPIRED	Translated from SSL alerts: certificate_expired	NO_PERMISSION BAD_PARAM
0x56422226	BAD_RECORD_MAC	Translated from SSL alerts: bad_record_mac	NO_PERMISSION
0x56422227	HANDSHAKE_FAILURE	Translated from SSL alerts: handshake_failure	NO_PERMISSION BAD_INV_ORDER

PKCS12 Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422251	P12_GENERAL_ERROR	Unknown error when processing PKCS12 data	BAD_PARAM NO_RESOURCES
0x56422252	P12_INVALID_DATA_FORMAT	Corrupted PKCS12 data	BAD_PARAM
0x56422253	P12_INVALID_PK_FORMAT	The private key retrieved from PKCS12 data is of unsupported format	BAD_PARAM
0x56422255	P12_MISSING_DATA	The PKCS12 data does not contain the required item	BAD_PARAM

General Security Policies (GSP) Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422271	SERVER_REQUIRES_SECURE_CONNECTIONS	Secure transport is not available in this client while the server requires it to connect using a secure connection	NO_PERMISSION
0x56422272	SECURITY_CURRENT_UNAVAILABLE	SSL session is not available, possible causes: not in a request context, not an SSL connection or invalid object reference (on client side)	BAD_OPERATION
0x56422273	NO_POSSIBLE_CONNECTION	All available connections to the server do not meet the security requirements set up on this client	NO_PERMISSION
0x56422274	SERVER_REQUIRES_TRANSPORT_IDENTITY	The server requires transport identity while this client does not have any	NO_PERMISSION

Common Secure Interoperable (CSI) Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422300	NO_IDENTITY	The server requires at least an identity while this client does not send any	NO_PERMISSION
0x56422301	BAD_SAS_DISC	This client sends SAS (Security Attribute Service, see OMG-CSI) that the server fails to interpret	NO_PERMISSION BAD_PARAM
0x56422302	UP_IDENTITY_REQD	The server requires a service context based U/P identity while this client does not send any	NO_PERMISSION
0x56422303	NO_CONTEXT	This is the minor code of the NO_PERMISSION exception thrown back to the client carrying ContextError with major = 4 and minor = 1 for 'no context' semantic as per OMG-CSI	NO_PERMISSION TRANSIENT
0x56422304	CONFLICTING_EVIDENCE	This is the minor code of the NO_PERMISSION exception thrown back to the client carrying ContextError with major = 3 and minor = 1 for 'conflicting evidence' semantic as per OMG-CSI	NO_PERMISSION TRANSIENT
0x56422305	ASSERTION_UNAUTHORIZED	This is the end-tier that is not supposed to make another call to the next tier on behalf of the caller. Please see vbroker.security.supportIdentityAssertion=true/false (default=true)	NO_PERMISSION

Authentication Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422400	LOGIN_FAILED	This is the minor code of the NO_PERMISSION exception thrown back to the client carrying ContextError with major = 1 and minor = 1 for 'invalid evidence' semantic as per OMG-CSI	NO_PERMISSION BAD_PARAM

Authorization Errors

Error Minor code	Error Identifier (ERROR_ID)	Error Description	Associated Corba Exception
0x56422500	FAILED_AUTHORIZATION	The caller has insufficient privileges required to perform the action	NO_PERMISSION
0x56422501	INVALID_ROLE	The required run-as role is not configured on this system	NO_PERMISSION
0x56422502	EMPTY_ALIAS_FOR_ROLE	The required alias for the specified run-as role is not configured on this system	NO_PERMISSION
0x56422503	ACCESSED_BY_UNKNOWN_USER	Access is denied because the caller is unknown. Caller is either null or anonymous	NO_PERMISSION
0x56422504	ACCESSED_BY_UNAUTHENTICATED_USER	Access is denied because the caller is not trusted	NO_PERMISSION

B

Appendix

Basic LoginModule

This LoginModule uses a proprietary schema to store and retrieve the user information. It uses the standard JDBC to store its data in any relational database. This module also supports the proprietary schema used by the Tomcat JDBC realm.

```
realm-name {
   com.borland.security.provider.authn.BasicLoginModule authentication-requirements-flag
   DRIVER=driver-name
   URL=database-URL
   TYPE=basic|tomcat
   LOGINUSERID=user-name
   LOGINPASSWORD=password
   [USERTABLE=user-table-name]
   [GROUPTABLE=group-table-name]
   [GROUPNAMEFIELD=group-name-field-of-GROUPTABLE]
   [PASSWORDFIELD=field-name]
   [USERNAMEFIELDINUSERTABLE=field-name]
   [USERNAMEFIELDINGROUPTABLE=field-name]
   [DIGEST=digest-name]
};
```

The elements in square brackets (“[..]”) are used only when authenticating to the Tomcat Realm, where they would be required. Otherwise, the remaining properties are sufficient.

Property	Description
DRIVER	The fully-qualified class name of the database driver to be used with the password backend. For example, `com.borland.datastore.jdbc.DataStoreDriver`
URL	The fully-qualified URL of the database used for the realm.
TYPE	The schema to use for this realm. This LoginModule supports the schema used by the Tomcat JDBC realm and can be made to use that schema. Set this to “TOMCAT” to use the Tomcat schema. Set this to “basic” to use the Borland schema. **Note:** If this property is set to “TOMCAT,” all other properties in square braces (“[..]”) must also be set.
LOGINUSERID	Username needed to access the password backend database.
LOGINPASSWORD	Password needed to access the password backend database.

Property	Description
[USERTABLE]	Table name under which the username/password to be authenticated is stored.
[USERNAMEFIELDINUSER-TABLE]	The field name in USERTABLE where the userID can be read.
[USERNAMEFIELDIN-GROUPTABLE]	The field name in GROUPTABLE where the userID can be read, different from that in the USERTABLE.
[PASSWORDFIELD]	The field name in USERTABLE containing the password for the username to be authenticated.
[GROUPTABLE]	The table name under where the group information for the user is stored. When TYPE is set to "TOMCAT," the attribute represented by entries in this table are treated as roles rather than groups.
[GROUPNAMEFIELD]	Name of the field in GROUPTABLE containing the group name to be associated with the user. When the TYPE is set to "TOMCAT," the attribute represented by entries in this table are treated as roles rather than groups.
[DIGEST]	The algorithm to use for digesting the password. This defaults to SHA under basic circumstances, but defaults to MD5 when TYPE is set to "TOMCAT".

```
Premium {
   com.borland.security.provider.authn.BasicLoginModule required
   DRIVER="com.borland.datastore.jdbc.DataStoreDriver"
   URL="jdbc:borland:dslocal:/Security/java/prod/userauthinfo1.jds"
   Realm="Basic"
   LOGINUSERID="CreateTx"
   LOGINPASSWORD="";
};
```

Since password should never be stored in the clear text, VisiSecure always performs digest on the password and stores the result in the database. The `digesttype` option defines the digest algorithm for this. By default, an SHA algorithm is used for basic-typed schema, while MD5 is used for tomcat-typed schema. You can change it by including and setting a `digesttype` option. In the case where the corresponding digest type engine cannot be found by the JVM, SHA is used instead. If an SHA engine cannot be found either, the authentication will always fail.

JDBC LoginModule

This LoginModule uses a standard JDBC database interface for authentication.

```
realm-name {
   com.borland.security.provider.authn.JDBCLoginModule authentication-requirements-flag
   DRIVER=driver-name
   URL=database-URL
   [DBTYPE=type]
   USERTABLE=user-table-name
   USERNAMEFIELD=user-name-field-of-USERTABLE
   ROLETABLE=role-table-name
   ROLENAMEFIELD=field-name
   USERNAMEFIELDINROLETABLE=field-name
};
```

Property	Description
DRIVER	Fully-qualified class name of the database driver to be used with the realm. For example, `com.borland.datastore.jdbc.DataStoreDriver`
URL	Fully-qualified URL of the database used for the password backend.

Property	Description
[DBTYPE= ORACLE|SYBASE|SQLSERVER| INTERBASE]	Supported database types. If this option is specified, the table information is preconfigured and need not be specified. The username/password still need to be specified to allow access to the system tables.
USERTABLE	Table name under where the database stores users.
USERNAMEFIELD	The field name in `USERTABLE` containing the usernames.
ROLETABLE	Table name where the database stores the roles of users.
ROLENAMEFIELD	Field name in `ROLETABLE` where role information is stored.
USERNAMEFIELDINROLE-TABLE	The username field name in the `ROLETABLE`.
USERNAME	The username needed to access the password backend database.
PASSWORD	The password needed to access the password backend database.

```
LIMS {
   com.borland.security.provider.authn.JDBCLoginModule required
   DRIVER="com.borland.datastore.jdbc.DataStoreDriver"
   URL="jdbc:borland:dslocal:/Security/java/prod/userauthinfo.jds"
   USERTABLE=myUserTable
   USERNAMEFIELD=userNames
   ROLETABLE=myRoles
   ROLENAMEFIELD=roleNames
   USERNAMEFIELDINROLETABLE=userRole
   USERNAME="\n"
   PASSWORD="\n";
};
```

LDAP LoginModule

Similar to the JDBC LoginModule, but using LDAP as its authentication backend.

```
realm-name {
   com.borland.security.provider.authn.LDAPLoginModule authentication-requirements-flag
   INITIALCONTEXTFACTORY=connection-factory-name
   PROVIDERURL=database-URL
   SEARCHBASE=search-start-point
   USERATTRIBUTES=attribute1, attribute2, ...
   USERNAMEATTRIBUTE=attribute
   QUERY=dynamic-query
};
```

Property	Description
INITIALCONTEXTFACTORY	The InitialContextFactory class that is used by JNDI to bind to LDAP.
PROVIDERURL	The URL to the LDAP server of the form `ldap://<servername>:<port>`.
SEARCHBASE	The search base for the Directory to lookup.
USERATTRIBUTES	This option controls the attributes that are retrieved for a given user. This is a comma separated list of attributes that will be retrieved and stored for an authenticated user. These attributes can then be used in the authorization rules to determine whether a user belongs to a given role.

Property	Description
USERNAMEATTRIBUTE	This attribute represents what the user types in as the username. If set to `uid`, it would allow users to type their `uid` when asked for a username. If set to `mail`, it would allow the user to type their email when asked for a user name. When set to `DN`, the user will types their full `DN` to authenticate themselves.

QUERY	The Query options provides a mechanism to dynamically query the LDAP for other information and represent the results as attributes. For example, a user can be a member of a set of groups. It is useful to extract this information as the `GROUP` attribute so that it can be used in rules in the authorization domain. To achieve this, you can specify a query. Queries are of the format:
	`query.<suffix>="<attrname>=<ldap filter>";`
	The `suffix` can be anything that uniquely identifies this entry and there can be any number of queries specified. To insert the user's DN as part of the query, you should use `{0}`. The `LDAPLoginModule` will then replace the `{0}` with the actual DN of the user. For example, to query groups and store the results in the `GROUP` attribute, you say:
	`query.1="GROUP=(&(ou=groups)(uniquemember={0}))";`
	This will select all the groups (whose `ou` attribute has the value groups) that the user belongs to whose `uniquemember` attribute contains the user's DN, then stores the CN of the objects returned as the result as the values for the `GROUP` attribute for that user. If the attribute name specified is `ROLE`, then this attribute's treatment is exactly like that of the `JDBCLoginModule`. This mechanism can be used to store user roles in LDAP.

Host LoginModule

The HostLoginModule is used to authenticate to a UNIX or NT-based network.

```
realm-name {
   com.borland.security.provider.authn.HostLoginModule authentication-requirements-flag;
};
```

No additional properties are necessary for the Host LoginModule.

```
Snoopy {
   com.borland.security.provider.authn.HostLoginModule required;
};
```

Shadow password for the Host LoginModule

On UNIX platforms:

The HostLoginModule shipped with VisiSecure for UNIX platforms utilizes simple APIs that is uniform on most UNIX platforms. This is defined in the POSIX standard header file `pwd.h`. Advanced shadow password APIs are available for deployments that demand higher security measures. However, one problem associated with this is that the process calling the APIs must run as root. Since the APIs are not in POSIX standard, the login module code is less portable.

To write your own custom login module, refer to the ‘customlogin’ example inside VisiSecure example folder. You may then incorporate shadow password APIs in your custom login module.

These APIs are available in the system header file: ‘shadow.h’.

Please consult your system manual to find out more about them.

From the user's perspective, as already indicated above, any VisiBroker application (client/server) configured with an authentication realm, employing such a login module, must be invoked with root (or SUID root) system-level privileges."

Creating user database for basic login module

As a first step, we'll create/configure our database to store users and roles. Borland provides the 'userdbadmin' tool (run from the command line) to auto-create required tables, create groups and associate users with groups.

We'll use JDataStore for this example; though any backend - like Oracle, DB2, Sybase, MS SQL Server, etc. can be used. A sample command is shown below. For JDataStore, the command is to be run from the command prompt when the current working directory is:

```
$BES/var/servers/[server_name]
  userdbadmin
    -create
    -db jdbc:borland:dslocal:adm/security/mydb.jds
    -driver com.borland.datastore.jdbc.DataStoreDriver
    -user admin
    -password admin
    -interactive
>addgroups accountant
  >addgroups clerk
  >adduser krish krishpwd accountant
  >adduser john johnpwd accountant
  >adduser bill billpwd clerk
  >adduser scott scottpwd clerk
  >quit
```

The above commands typed at the ">" prompt creates two groups 'accountant' and 'clerk' in the database. Two users with usernames krish and john are in an accountant role; while bill and scott are in the role of clerks. (Type 'help' at the ">" prompt for a list on available commands).

Using userdbadmin tool

The `userdbadmin` is a command-line tool that can be used to create and manage user databases for the BasicLoginModule. The `userdbadmin` uses a properietary schema and can be pointed at any database. Using this tool, you can administer users who can be authenticated uing Basic login modules. Though the tool and BasicLoginModule work using various JDBC databases, it is still recommended that you use Borland JDataStore that is shipped with the VisiBroker.

To facilitate the use of popular databases, the `userdbadmin` tool comes pre-configured to recognize database urls (keyed of the vendor and configure itself to use the appropriate drivers.

If you wish to change that, you may override by specifying the driver information.

If you don't provide driver information, the userdbadmin does not recognize the database and it will prompt for this information. Once it has successfully acquired this information, it will write this information into a file called `.userdbadmin.config` in the directory corresponding to the `user.home` system property or to the file specified by the -config command line option.

Future users of `userdbadmin` will read the config file from either the `user.home` directory or from the file specified by the `-config` option and will recognize the new database configuration, so you don't have to type the driver information every time.

Creating a new database

To create a new database, use the following commands below.

```
Usage: userdbadmin [<driver options>] [<userdbadmin options>] [command]
```

The example below creates a new database namely `mydb.jds`.

```
prompt> userdbadmin -db jdbc:borland:dslocal:mydb.jds -driver
com.borland.datastore.jdbc.DataStoreDriver -user administrator -password b0rlanD -create
```

The `username/password` that you supply in the command line above is used by JDataStore to protect the database as well as to access to the database subsequently.

Note: The username/password is for JDataStore itself. This has nothing to do with the usernames and passwords that you may want to store in the database later.

```
UserdbAdminTool: Creating database jdbc:borland:dslocal:mydb.jds
JDataStore: Developer's License (no connection limit)
JDataStore: Copyright (c) 1996-2004 Borland Software Corporation. All rights reserved.
JDataStore: License for JDataStore development only - not for redistribution
JDataStore: Registered to:
JDataStore: JDataStore
JDataStore: Developer's license with unlimited connections
Password digest algorithm is SHA1
UserdbAdminTool: Created Database Schema
prompt>
```

After the execution of the command, in your current directory, a new set of JDataStore database physical files will be created as follows:

```
mydb.jds
mydb_LOGA_0000000000
mydb_LOGA_ANCHOR
mydb_STATUS_0000000000
```

Creating groups and associating users with groups

Launch `userdbadmin` tool in an interactive mode with the created database. The interactive mode helps you to issue multiple commands.

To launch `userdbadmin` tool in an interactive mode with the created database, enter the command as given below at the command prompt.

```
prompt> userdbadmin -db jdbc:borland:dslocal:mydb.jds -driver
com.borland.datastore.jdbc.DataStoreDriver -user administrator -password b0rlanD -
interactive

JDataStore:  Developer's License (no connection limit)
JDataStore:  Copyright (c) 1996-2004 Borland Software Corporation.
All rights reserved.
JDataStore:  License for JDataStore development only - not for redistribution
JDataStore:  Registered to:
JDataStore:  JDataStore
JDataStore:  Developer's license with unlimited connections
Password digest algorithm is SHA1
Enter "quit" to quit.
>
```

Note: You are inside `userdbadmin` interactive mode that is waiting for you to type commands at its '>' prompt.

Adding new users

To add a new username with password and make the user member of the group(s), type the following below in the command line.

Example 1:

```
> adduser krish krishpwd accountant
```

Example 2:

```
> adduser bill billpwd clerk
```

In example 1, you have added a `user` whose name is `Krish` and password `krishpwd` and added him as the user member of the group called `accountant`.

In example 2, you have added a `user` whose name is `bill` and password `billpwd` and added him as the user member of the group called `clerk`.

Listing exising users in the database

To list existing users in the database, type `'listusers'` in the command line to list all the users and their groups:

```
> listusers
bill: [ clerk ]
john: [ accountant ]
krish: [ accountant ]
```

Listing all groups in the database

To list all groups and their membership, enter `listgroup` at the command prompt.

```
> listgroups
clerk: [ bill ]
accountant: [ krish john ]
```

Create new groups and check using listgroups

To create new groups and their membership, enter `addgroup` at the command prompt.

```
> addgroups dba admin
```

You can check the newly added group by running the command listgroup. The newly added group dba would be listed.

```
> listgroups
dba: [ ]
admin: [ ]
clerk: [ bill ]
```

Assign groups to existing users

To assign user krish to group dba and group admin, enter the following command at the prompt

```
> joingroups krish dba admin
```

You can check the newly added user by running the command listusers. The newly added group dba would be listed.

```
> listusers
bill: [ clerk ]
john: [ accountant ]
krish: [ accountant dba admin ]
```

Remove group from database

To remove group accountant from the database permanently, enter the following command `removegroups` at the prompt.

```
> removegroups accountant
```

You can check the newly removed group by running the command `listusers`. The newly removed group accountant will not be listed.

```
> listusers
bill: [ clerk ]
john: [ ]
```

```
krish: [ dba admin ]
```

Add a new user without any group

You can add a new user without adding him to any specific group. enter the command `adduser` at the command prompt

```
> adduser jack jackpwd
```

You can check the newly added user by running the command `listusers`. The newly added group dba would be listed.

```
> listusers
bill: [ clerk ]
jack: [ ]
john: [ ]
krish: [ dba admin ]
```

Remove group admin from user

To remove the group from the user, enter the command `leavegroup` at the command prompt.

```
> leavegroups krish admin
```

You can check the check the newly removed user by running the command `listusers`. The newly removed user would be listed.

```
> listusers
bill: [ clerk ]
jack: [ ]
john: [ ]
krish: [ dba ]
```

Remove user from the group

To remove the user from the group, enter the command `leave group` at the command prompt.

```
> removeuser bill
```

You can check the check the newly removed user by running the command `listusers`. The newly removed user would be listed.

```
> listusers
jack: [ ]
john: [ ]
krish: [ dba ]
```

Exiting the userdbadmin program

To exit out of the `userdbadmin` program, enter the command `quit` at the command.

```
> quit
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

Index

T

Index

U

V

W