VisiBroker Security Guide

V8.5.9

Table of Contents

Introduction to VisiBroker 10
- Accessing VisiBroker online help topics in the standalone Help Viewer 10
- Accessing VisiBroker online help topics from within a VisiBroker GUI tool 11

Getting Started with VisiSecure 12
- VisiSecure Overview 12
  - VisiSecure design flexibility 13
  - Pluggability 13
  - VisiSecure for Java 13
  - VisiSecure for C++ 14
- Basic security model 14
  - Authentication realm (user domain) 15
  - Resource domain 16
  - Authorization domain 16
- Distributed transmission 16

Authentication 18
- Managing authentication with JAAS 18
  - Basics of JAAS concepts 18
  - Subjects 18
  - Principals 19
  - Credentials 20
  - Authentication and pluggability 21
- Authentication mechanisms and LoginModules 22
- Identity, trust and authentication 22
  - Relationship between trust and authentication 22
  - Identities 23
- Configuring authentication 24
  - Authentication property settings 25
  - Formatted target 26
  - Setting the config file for client authentication 26

Setting up authentication realms 27
Different types of Authentication 28
Servers 28
Clients 28
Authentication Mechanisms 29
GSSUP mechanism 29
Certificate mechanism 32
Creating LoginModules 42
LoginContext class and LoginModule interface 42
Authentication and stacked LoginModules 43
Associating a LoginModule with a realm 44
Syntax of a realm entry 45
LoginModules 47
Using a Vault 48
Creating a Vault 49
Certificate Revocation List (CRL) and revoked certificate serial numbers 52
Support for MS-CAPI 53
VisiBroker C++ Only 54
VisiBroker Java Only 54
Authorization 55
Access Control 55
Access Control List 55
Roles-based access control 55
Pluggable Authorization 56
Configuring authorization using the rolemap file 57
What is a rolemap 57
Syntax of role DB 57
Modifying the authorization rolemap file 58
Authorization domains 61
Specifying names to authorization domain 61

Configuring authorization domains to run-as alias 62
Setting up authorization for CORBA objects 63
Setting up the name 64
Setting up default access 65
Configuring authorization requirements 65
Java Example: Authorization Using a Vault 68
To launch run-as alias 75
Run-as mapping 76
Using a vault for a domain 77
Context Propagation 77
Identity assertions 78
Asserting identity of the caller 79
Trusting Assertions 83
Temporary privileges 84
Secure Transportation 85
Encryption 85
Public-key encryption 85
Asymmetric encryption 86
Symmetric encryption 86
Certificates and Certificate Authority 87
Generating a private key and certificate request 88
Digital signatures 89
Enabling SSL 89
Setting the level of encryption 89
Enabling Security 95
Enabling SSL/TLS 95
Setting the Log Level 95
Using IIOP/HTTPS 97
Browser considerations 97

Microsoft Internet Explorer 98
Quality of Protection 99
Setting properties and QoP 99
Configuring Quality of Protection(QoP) 100
Configuring QoP for the server 100
Configuring QoP for the client 102
Configuring Quality of Protection (QoP) parameters 105
Creating custom plugins 107
LoginModules 107
CallbackHandlers 110
Authorization Service Providers 111
Example 112
Trust Providers 122
Creating Secure CORBA Applications Using Java 123
Steps to secure clients and servers 123
Step One: Providing an identity 123
Clients 123
cServers 124
Step Two: Setting properties and Quality of Protection (QoP) 126
Step Three: Setting up Trust 126
Step Four: Setting up the Pseudo-Random Number Generator 127
Step Five: If necessary, set up identity assertion 127
Examining SSL related information 127
Clients 128
Servers 128
SSL Example 128
Creating Secure CORBA Applications Using C++ 130
Step One: Providing an identity 130
Clients 130
Servers 131
Step Two: Setting properties and Quality of Protection (QoP) 133

Step Three: Setting up Trust 133
Step Four: If necessary, set up identity assertion 134
Security configuration while setting up a server engine 134
Examining SSL related information 135
SSL example 135
Using properties to install certificates, private key and trustpoints 135
Using initializers to install certificates, private key, trustpoints and CRL 136
Using APIs with Security aware applications: SecureServer and SecureClient 137
Using APIs with pkcs12Server 138
Secure Object URLs 139
corbaloc:ssliop Object URLs 139
corbaname:ssliop Object URLs 140
Quality of Protection for Secure Object URLs 141
Security Properties for Java 143
SSL/TLS Server Connection Manager properties 169
Security Properties for C++ 172
SSL Server Connection Manager properties 185
Security SPI for C++ 188
Plugin Mechanism and SPIs 188
Providers 190
Providers and exceptions 191
vbsec::LoginModule 192
Include File 192
Methods 192
vbsec::CallbackHandler 194
Include file 194
Methods 194
vbsec::IdentityAdapter 194
IdentityAdapters included with the VisiSecure 195
Methods 195

- vbsec::MechanismAdapter 198
vbsec::AuthenticationMechanisms 199
- Credential-related methods 200
- Context-related methods 201
vbsec::Target 203
- Methods 204
vbsec::AuthorizationServicesProvider 205
- Methods 205
vbsec::Resource 207
- Methods 207
vbsec::Privileges 207
- Constructors 208
- Methods 208
vbsec::AttributeCodec 210
- Methods 211
vbsec::Permission 214
- Include file 214
- Methods 214
vbsec::PermissionCollection 216
- Include file 216
- Methods 216
vbsec::RolePermission 216
- Constructors 216
- Methods 217
vbsec::TrustProvider 218
- Methods 218
vbsec::InitOptions 220
- Include file 220
- Data Members 220
vbsec::SimpleLogger 221
- Include file 221

Methods 221
VisiSecure for C++ APIs 223
General API 223
class vbsec::Current 223
class vbsec::Context 225
class vbsec::Principal 229
class vbsec::Credential 230
class vbsec::Subject 230
class vbsec::Wallet 232
class vbsec::WalletFactory 233
SSL API 235
class vbsec::SSLSession 235
class vbsec::VBSSLContext 237
class ssl::CipherSuiteInfo 238
class CipherSuiteName 238
class vbsec::SecureSocketProvider 239
class ssl::Current 240
Certificate API 244
class vbsec::CertificateFactory 244
class CORBAsec::X509Cert 247
class CORBAsec::X509CertExtension 250
QoP API 251
class vbsec::ServerConfigImpl 251
class ServerQoPPolicyImpl 252
class vbsec::ClientConfigImpl 253
class vbsec::ClientQoPPolicyImpl 253
Authorization API 254
class csiv2::AccessPolicyManager 254
class csiv2::ObjectAccessPolicy 255
VisiSecure Error Codes 257
Modifying minor codes in C++ 257

- Modifying minor codes in Java 257
- General Errors 258
- PKI Errors 258
- SSL Errors 259
- PKCS#12 Errors 260
- General Security Policies (GSP) Errors 261
- Common Secure Interoperable (CSI) Errors 262
- Authentication Errors 263
- Authorization Errors 264

Login Modules 265

- Basic LoginModule 265
- JDBC LoginModule 267
- LDAP LoginModule 269
- Host LoginModule 271
  - UNIX shadow password for Host LoginModule 272
- Creating a user database for the basic login module 272
  - Using the userdbadmin tool 273

Notices 279

- Copyright 279
- Trademarks 279
- Examples 279
- License agreement 279
- Corporate information 280
- Contacting Technical Support 280
- Country and Toll-free telephone number 280

Introduction to VisiBroker

VisiBroker is a set of services and tools that enables you to build, deploy, and manage distributed enterprise applications in your corporate environment. These applications provide dynamic content by using JSP, servlets, and Enterprise Java Bean (EJB) technologies.

Accessing VisiBroker online help topics in the standalone Help Viewer

To access the online help through the standalone Help Viewer on a machine where the product is installed, use one of the following methods:

Windows

- Click **Start** > **Programs** > **VisiBroker** > **Help Topics**

  or:

- Open a command prompt and go to the product installation `\bin` directory, then enter the following command:

  `help`

UNIX

Open a command shell and go to the product installation `/bin` directory, then enter the command:

`help`

Tip

During installation on UNIX systems, the default is to not include an entry for `bin` in your `PATH`. If you did not choose the custom install option and modify the default for `PATH` entry, and you do not have an entry for current directory in your `PATH`, use `./help` to start the help viewer.

Accessing VisiBroker online help topics from within a VisiBroker GUI tool

To access the online help from within a VisiBroker GUI tool, click **Help > Help Topics**.

The Help menu also contains shortcuts to specific documents within the online help. When you select one of these shortcuts, the Help Topics viewer is launched and the item selected from the Help menu is displayed.

Getting Started with VisiSecure

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

Sensitive information routinely passes over Internet connections between web browsers and commercial web servers; credit card numbers and bank balances are 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, not an impostor 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 BDOC. VisiSecure lets you establish secure connections between clients and servers.

VisiSecure design flexibility

VisiSecure is designed to work with a variety of application architectures, so that it can support many different current and future architectures. However, while VisiSecure represents a powerful security architecture, it alone cannot fully protect your servers. 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 for Java 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, the VisiSecure feature set exceeds the J2EE 1.3 security requirements.

VisiSecure for Java features

VisiSecure for Java 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 options additional to 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 Rocket Software.

- **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 for Java utilizes standardized Transport Layer Security (TLS) which has evolved from the older SSL standards. Note that Java only supports the TLS versions provided by its underlying security provider (such as JSSE).

VisiSecure for C++

VisiSecure for C++ offers similar features to 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 for C++ utilizes standardized Transport Layer Security (TLS) which has evolved from the older SSL standards. This release includes support for TLSv1.3, TLSv1.2, TLSv1.1 and TLSv1.0.

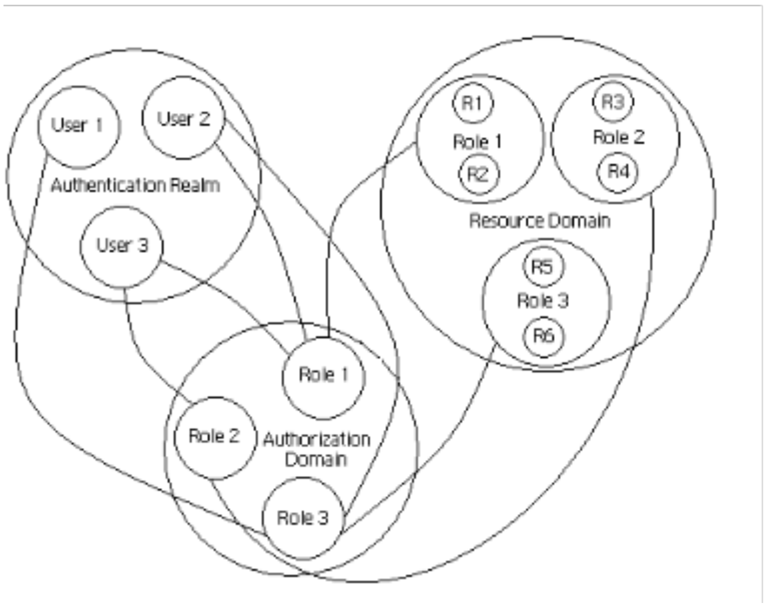
Basic security model

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):** simply a database of users. Each authentication realm describes a set of users and their associated credentials and Privileges attributes.
- **Resource Domain:** represents a collection of resources of a single application. The application developer defines the access control policies for access to resources in the application.

- **Authorization Domain:** defines the set of rules that determines whether an access attempt to a particular resource is allowed.

  The following figure displays the relationship among these domains.



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, simply described, 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 that it uses (e.g. LoginModules), and by a set of configuration options that 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 an authentication realm, see Authentication.

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 endpoint that is exposed by a server is essentially a resource.

The application developer defines access control policies for access to 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, access is denied. For more information, 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. Because 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

Authentication

The first layer of security protection for any system is authentication (as well as identity representation). This layer defines the process of verifying the entities are 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.

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. The three essential concepts of the framework are *Subject*, *Principal*, and *Credential*.

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 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, as illustrated by 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 Principal 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 and public-key certificates. 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 so on, while private Credentials are private keys, encryption keys, sensitive passwords, and so on. Consider the following methods for `class Subject`:

Java

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

C++

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

No additional permissions would be necessary to retrieve the public Credentials from the Subject, but the situation changes when trying to retrieve *private* Credentials with following APIs:

Java

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

C++

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

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

Public Credentials are 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, consult the JAAS Specification from Oracle.

Authentication and pluggability

Within the JAAS framework, the logon service separates configuration from implementation. A low-level system programming interface called the LoginModule provides an anchor point for pluggable security modules.

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 Authentication Mechanism and defines a set of classes for doing authentication/identification in a distributed environment.

Authentication mechanisms and LoginModules

An *Authentication Mechanism* represents the encoding/decoding 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.

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

Identity, trust and authentication

Relationship between trust and authentication

Authentication is a process of verifying an identity. When verification is successful, 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 identity assertion.

Assertion works as follows:

1. Identity A is successfully authenticated and is therefore trusted.
2. Later, identity A asserts identity B.
3. Identity B is now trusted, although we never directly verify its identity. This is because the system trusts A and trusts 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 provide a Java keystore where certificate entries are trusted using standard Java properties.

C++

For VisiBroker for C++ users, a set of APIs that allow trustpoints (trusted certificates) to be configured is provided. 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/TLS 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, 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 to be able to identify themselves in two scenarios:

- When using SSL/TLS for transport layer security, the server typically needs to identify itself to the client.
- 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, however, where the client process does not have any information on the realm against which it needs to authenticate. 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. For example, the retry count can be modified by setting `vbroker.security.authentication.retryCount`. Its default value is `3`. The security properties including those for authentication are listed and described in 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, 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, see Associating a LoginModule with a realm.

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

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

This defines a realm called "customrealm", which requires the use of the `CustomLoginModule`. Each realm entry has a particular syntax that must be followed. For more information on realm syntax, see Syntax of a realm entry.

A login configuration contains the following information:

- 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 a description of the valid values for Flag and their respective semantics, see Syntax of a realm entry.

For information on LoginModules, see LoginModules.

Authentication property settings

The authentication policy, whether it is for server or client authentication and whether it is to use public-key certificates or passwords, 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 LoginModules necessary for authentication to the various realms to which this process needs to authenticate.

For example:

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

In `<config-file>`, set up 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:

`<Authentication Mechanism>#<Authentication Target>`

This format is called *Formatted target*.

Formatted target

Formatted target is the canonical representation of an 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 the GSSUP Authentication Mechanism. GSSUP is the assumed default type when no other Authentication Mechanism is specified.

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 mechanisms, see Authentication Mechanisms.

Setting the config file for client authentication

Each process uses its own configuration file to configure the set of authentication realms that the system is to support 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=<physical file name for myconfig>
vbroker.security.authConfig.yourconfig=<physical file name for yourconfig>
vbroker.security.authConfig.hisconfig=<physical file name for hisconfig>
vbroker.security.authConfig.herconfig=<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 is 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.

Follow these steps to setup the authentication realm:

1. Create an authentication configuration file containing one or more realms. For information on Creating configuration file, see Configuring authentication.
2. Use the property `vbroker.security.authentication.config` to enable use of the configuration file by the runtime.

   `vbroker.security.authentication.config=<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 VisiSecure to refer to this entry. The entry specifies the LoginModule to be used for 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 further information on setting up the configuration file for client and server, refer to the basic authentication example in the `<install_dir>\examples\vbroker\security\basic` folder. That example has all the basic settings needed to secure an application.

For more information on the authentication realm (user domain), see Authentication realm (user domain).

Different types of Authentication

With the VisiBroker implementation of JAAS, you can set up different mechanisms for 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. Whether it is server or client authentication, and whether it is done using public-key certificates or passwords, 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.

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

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

The client can authenticate itself if the server's realm is known up-front. Even in such cases the server will authenticate again.

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

Authentication Mechanisms

An *Authentication Mechanism* represents the encoding/decoding 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 Mechanism 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 Rocket Software. This is because the mechanism implementation needs to translate the information provided by a LoginModule to a specific format for transmitting 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, the “GSSUP#” prefix can be omitted from the target name. GSSUP is the default Authentication 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 methods are:

- 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* ctx = 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>);
  ctx->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, see LoginModules.

```
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 uses certificates for authentication. This mechanism is different from GSSUP; certificates are used instead of username/password, and these identities are used at the SSL/TLS 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.

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 table 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/TLS connections.

Target	Description
`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/TLS connections.
`Certificate #ALL`	Identifies this process in all roles. This identity is used in both of the above roles.

A process can have distinct client and server identities or a single identity that it uses for both client and server roles. 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`.

Typical uses of certificate-based authentication methods are:

- Certificate-based authentication using KeyStores and property settings
- Certificate-based authentication using APIs
- Certificate based authentication using APIs with pkcs12Server
- Certificate based authentication using Certificate wallet
- PKCS#12-based authentication using KeyStores
- PKCS#12-based authentication using APIs

Certificate-based authentication using KeyStores and property settings

This section demonstrates how to make a minimal SSL/TLS configuration on the simplest, security-unaware VisiBroker example such that client and server communicate using SSL/TLS to perform mutual PKI authentication. Executables from the `basic/bank_agent` example are re-used, to emphasize the fact that a non-security-aware application can be secured without requiring any change to source code.

1. Familiarize yourself with the C++ version of the simplest VisiBroker examples in the directory `basic/bank_agent`.
2. Copy over only the executables: `Server` and `Client` (`Server.exe` and `Client.exe` on Windows) to the current directory, `security/bank_ssl`.
3. Make sure that `osagent` is up and running as usual.

4. Launch the server using the command below:

   `prompt> ./Server -DORBpropStorage=cpp_server.properties -Dvbroker.orb.dynamicLibs=<VisiSecure shared library name>`

5. Launch the client using the command below:

   `prompt> ./Client -DORBpropStorage=cpp_client.properties -Dvbroker.orb.dynamicLibs=<VisiSecure shared library name>`

6. Open the property files `cpp_server.properties` and `cpp_client.properties` and notice how the certificates and private keys are installed using the wallet property set in the files.

   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
```

   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 descriptions of these properties, see Security Properties for C++.

7. Browse through the content of the subdirectories `identities` and `trustpoints` to learn how the directory wallet and trustpoints are structured.

Note

The VisiSecure shared library name depends on the platforms. For example:

- on win32, it is `vbsec.dll`.
- on Solaris 64 bit, it is `libvbsec64.so`.
- on HPUX 64 bit std build, it is `libvbsec64_p.sl`.

We recommend that you check your UNIX `<install_dir>/lib` directory or Windows `<install_dir>bin` directory.

Certificate-based authentication using APIs

1. Build the `security/bank_SSL` example (as described by `bank_ssl.html`) by executing the command:

   `nmake cpp` (for Windows)

   or:

   `make cpp` (for UNIX).

   A successful build creates the executables `SecureServer.exe` and `SecureClient.exe` on Windows, or `SecureServer` and `SecureClient` on UNIX.
2. Make sure `osagent` is up and running.
3. Launch the server using the command:

   ```
   prompt> ./SecureServer
   ```
4. Launch the client using the command:

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

   For example:

   ```
   prompt> ./SecureClient -Dvbroker.app.useCRL=true
   ```
6. Read and learn from `SecureServer.C` and `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) {
  // Wrap the b64 certificate chain in an ASN1ObjectList as
  // required by the certificate factory
  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 );
  }

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

Note

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

```
...
L = (CORBA::ULong) strlen(bank::privateKey);

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

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

const char* const sword = "frans";
L = (CORBA::ULong) strlen(sword);

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

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

The root of this chain is an 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 `CertificateWalletAPI`. This class also supports the PKCS#12 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 the VisiSecure API for handling a PKCS#12 file, a widely accepted storage format for certificates and private keys.

1. Build the `bank_ssl` example in the `<install_dir>/examples/vbroker/security` folder by executing the command:

   `nmake cpp` (for Windows)

   or :

   `make cpp` (for UNIX).

   When the build completes, the executable `pkcs12Server.exe` will have been created on Windows, or `pkcs12Server` on UNIX platforms.
2. Make sure `osagent` is up and running.
3. Launch the server using the command below:

   ```
   prompt> ./pkcs12Server frans.pfx frans
   ```
4. Launch the client using the command below:

   ```
   prompt> ./SecureClient
   ```
5. Launch the client using `-Dvbroker.app.useCRL=true`, and notice how the mutual SSL/TLS authentication fails and the client throws a `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 `PKCS#12` 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 );

  // Consult the certificate factory to convert the chain
  // into an X509CertList as required to create an SSLContext
  CORBAsec::X509CertList_var certchain =
      ssp->getCertificateFactory().importCertificateChain ( pkcs12bytes,
          sword );
...
```

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

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

if ( !certchain || !privatekey ) {
  cerr << "Fail to import certificates and private key from pkcs12 "
      << "file named: " << argv[1] << endl;
  exit(1);
}

// Consult the SecureSocketProvider to create an
// SSLContext from the chain and the private key.
vbsec::VBSSLContext* sslctx = ssp->createSSLContext( *certchain,
    *privatekey, sword );
...
```

Certificate based authentication using Certificate wallet

Use a certificate wallet as follows:

1. Create a new wallet

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

2. Get the security context:

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

3. Pass the wallet that was initially created in step :

```
ctx.login (wallet)
```

Setting wallet properties

You can set the transport identity for SSL using the properties below:

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

PKCS#12-based authentication using KeyStores

You can use the same APIs as discussed in Username/password authentication using APIs to login using `PKCS#12` 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 `PKCS#12` file. The property `javax.net.ssl.KeyStore` specifies the location of the `PKCS#12` file.

Specify the details of the `PKCS#12` KeyStore file as follows:

For C++:

```
vbroker.security.wallet.type=PKCS12:./identities/client.p12
vbroker.security.wallet.identity=client
vbroker.security.wallet.password=password
```

Note

The `vbroker.security.wallet.identity` property is ignored for a PKCS#12 keystore, but the property must be set.

For Java:

```
vbroker.security.wallet.type=PKCS12:./identities
vbroker.security.wallet.identity=client
vbroker.security.wallet.password=password
```

The VisiBroker for Java secure client then looks for a `client.p12` file in the `./identities` folder.

PKCS#12-based authentication using APIs

If you do not want to use KeyStores directly, you can import certificates and private keys using the `CertificateFactoryAPI`. This class also supports the `PKCS#12` 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 point 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 to plug-in under the current application.

Java

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

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.

Java

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

C++

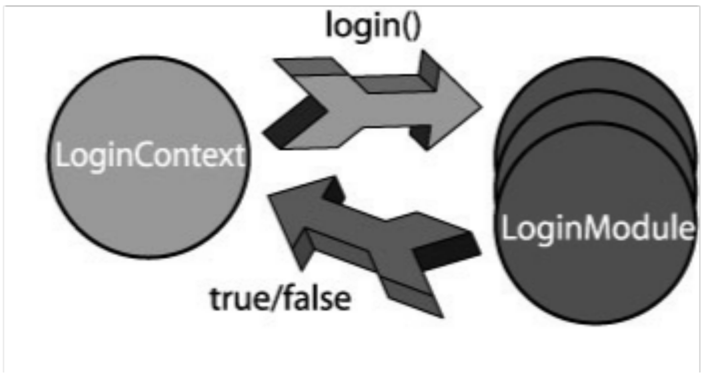
```
class LoginModule {
public:
  virtual bool login()=0;
  virtual bool logout()=0;
  virtual bool commit()=0;
  virtual bool abort()=0;
}
```

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

Authentication and stacked LoginModules

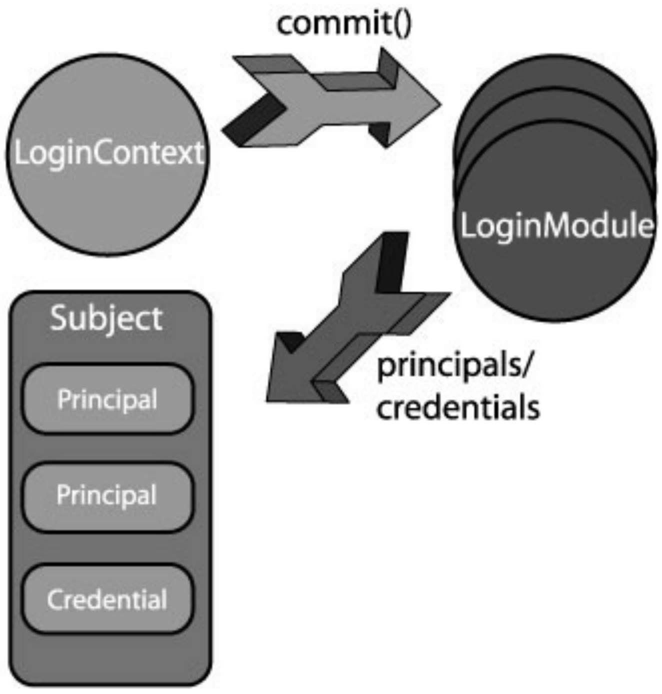
Authentication proceeds in two phases in order to assure that all stacked `LoginModules` succeed (or fail, otherwise).

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

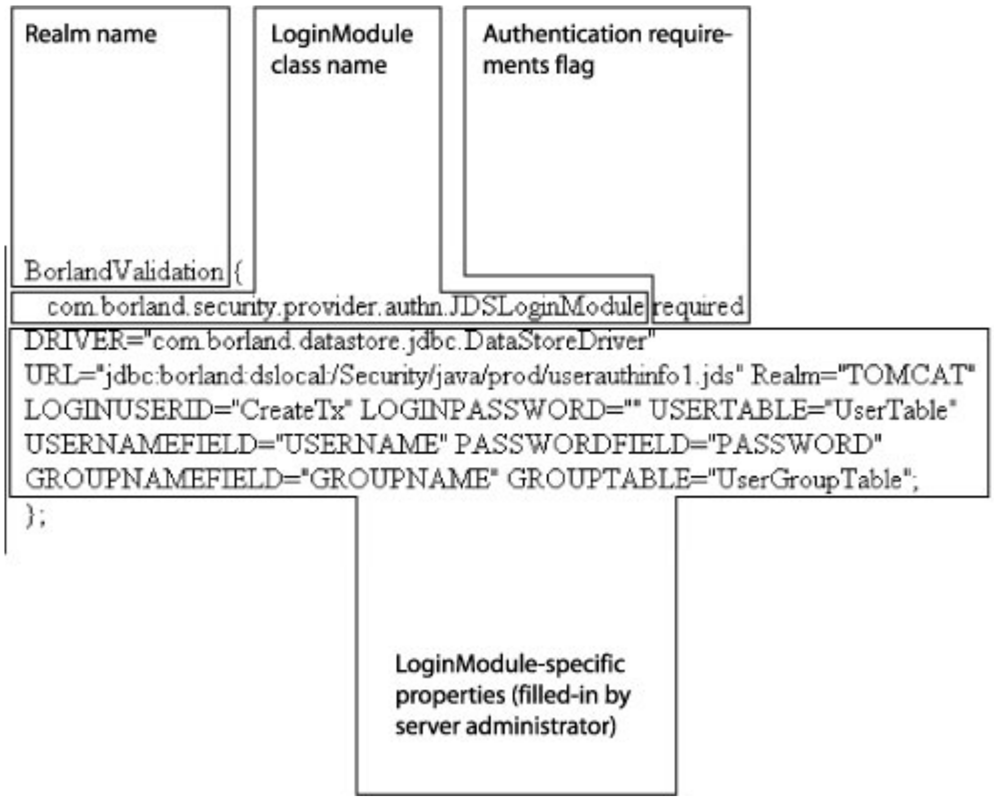
If either phase fails, the `LoginContext` calls `abort()` on each `LoginModule` and ends all authentication attempts.

Associating a LoginModule with a realm

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.


Realm name
LoginModule class name
Authentication requirements flag
LoginModule-specific properties (filled-in by server administrator)

```
BorlandValidation {
   com.borland.security.provider.authn.JDSLoginModule required
DRIVER="com.borland.datastore.jdbc.DataStoreDriver"
URL="jdbc:borland:dslocal:/Security/java/prod/userauthinfo1.jds" Realm="TOMCAT"
LOGINUSERID="CreateTx" LOGINPASSWORD="" USERTABLE="UserTable"
USERNAMEFIELD="USERNAME" PASSWORDFIELD="PASSWORD"
GROUPNAMEFIELD="GROUPNAME" GROUPTABLE="UserGroupTable";
};
```

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 set to `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 possible 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 provided LoginModule are described below.

LoginModules

Rocket Software 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 - similar to the JDBC LoginModule, but uses LDAP as its authentication back-end.
- Host LoginModule - for authentication to the operating system hosting the server. **This is the only LoginModule supported for C++.**

Using the callback handler to communicate with users

Sometimes the LoginModule needs to communicate with the user to obtain authentication information, for example asking the user to provide username and password. The LoginModule uses the callback handler for this purpose. Out of the box, VisiSecure provides command-line based callback handlers.

GUI-based callback handler

A GUI-based callback handler is provided in VisiSecure Java Edition for all login modules. To use the GUI login you must specify the correct callback handler, by setting the following property:

```
vbroker.security.authentication.callbackHandler=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 mentioned in Using the callback handler to communicate with users. 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 if a server is started as a service during host start-up, or from an automated script of some sort.

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

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. Its usage is as follows:

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

`<driver-options>` are optional, but 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.jaas-file>` points to the location of the `config.jaas` file containing the realms the identities in the vault will authenticate to.

`-vault <vault-name>` is the path to the vault to be generated. You can 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 require `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

This example uses VaultGen. Assume 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:

`<install_dir>/var/security/profiles/default/config.jaas`

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:

`<install_dir>/bin> ./vaultgen -config ../var/security/profiles/default/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:

`<install_dir>/bin> ./vaultgen -config ../var/security/profiles/default/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 the 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 settings needed to secure an application.

The API `generateVault` will take a file output stream and store 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 use username/password authentication of the client on the server and also for the server's self-authentication.

The different properties files (`server.properties`, `client.properties`) and the `server.config` files for the basic vault example can be found the `<install_dir>\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.

Generate the vault file

1. At the command prompt, enter the following in the server window:

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

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

   To run the server without providing authentication information:

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

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

2. To run the client, simply use the command:

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

   where `<vbsec library>` is:

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

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

3. The vaultfile uses this file information to login 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 (CRL) and revoked certificate serial numbers

C++ Only

When signed public key certificates are created by a Certificate Authority (CA), each certificate has an expiration 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 expiration, the Certificate Revocation List (CRL) feature is provided for VisiSecure for C++. For more information about Certificate Authorities (CA)s, see 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/TLS handshake communication.

The CRL files can be in either DER binary format or base 64 text format (PEM format). When an application adds a trusted certificate into a `VBSSLContext` instance, the corresponding 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 implemented as a CORBA octet sequence. See the `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 can be found 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 corresponding CRL.

!! notes - 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 the SSL/TLS handshake. If any of the peer certificates appears in
the CRLs, an exception will be thrown and the connection will be refused.
```

Support for MS-CAPI

Windows

On Windows systems, VisiBroker supports the Microsoft Cryptography API (CAPI).

VisiBroker C++ Only

Warning

VisiBroker for C++ has withdrawn support for use of keys and certificates held by the Microsoft certificate store.

As a result of updating to OpenSSL 3.0, VisiBroker for C++'s existing CAPI implementation has become unviable. This is because OpenSSL 1.1 supported CAPI via its Engine API. OpenSSL 3.0 has deprecated the Engine API (in favor of the Provider API).

Rocket Software intends to restore support for MS-CAPI in a future release.

Existing applications that use vbroker.security.useCAPI will find that the effective behavior is as if vbroker.security.useCAPI is set to `false`.

VisiBroker Java Only

When CAPI is fully enabled, it takes over the mechanism for some cryptographic operations, most notably generating RSA, DSA and ECDSA signatures. That means that private keys must be stored in Windows stores if CAPI is being used this way by a VisiBroker process. This is currently only supported on the client side.

CAPI support is automatically enabled in VisiBroker for Java.

The property vbroker.security.mscapiAliasFix was introduced to deal with a Java bug that has now been fixed in later Java version updates.

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 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. Rocket Software 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 has the necessary permissions 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 of 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

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, see 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 permissions that are granted access to certain resources.

The class `RolePermission` is defined to represent “role” as permission. 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 the `RolePermission` class, see vbsec::RolePermission.

The Authorization Service is tightly connected with the concept of the Authorization domain - each domain has exactly one Authorization Services Provider implementation. The Authorization domain is the bridge between the 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 that are 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 run time, this property is loaded by way of Java reflection.

Another important 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` is again consulted to return the alias that must be used to “run-as” in the context of the rules specified for this authorization domain.

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 the rules, and who are therefore associated with the corresponding role.

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 proceeds until one or none succeeds. 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 Assertion syntax.

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 the property `vbroker.security.domain.<authorization-domain>.rolemap_path`.

Modifying the 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 Assertion syntax.

For configuring the database to store users, credentials and attributes, refer to Using the userdbadmin tool.

Assertion syntax

There is 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, it may be prudent to look for only one or two attributes from the hierarchy in order to authorize a principal. VisiSecure's security hierarchy starts with GROUPs at the top, then branching out into ORGANIZATIONs (O) and ORGANIZATIONAL UNITS (OU), and finally settling 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 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.

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`. 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.

Specifying names to authorization domain

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 properties described in the following sections 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 each 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

The path of the Role DB file is associated with the authorization domain <`domain-name`>. Although this can be a relative path, Rocket Software 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.

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.

Note

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

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 the `server.properties` file. This property specifies the name of the run-as role. The value can either:

    - be *use-caller-identity* to have the caller principal itself as the principal identity for the run-as role, or
    - specify an alias for a principal to use as the run-as role name:

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

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 assertions made by peers by setting the property below to `true`:

   `vbroker.security.assertions.trust.all`

Setting up authorization for CORBA objects

Authorization in the CORBA environment allows only identities in specific roles for a given object to 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. Similarly, the J2EE roles and concepts therein are also used in the CORBA environment.

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,
      ::CORBA::OctetSequence& _adapter_id ) = 0;
}
```

Setting up the name

This interface returns the authorization domain from the `domain()` method and uses it to 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. Implement 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 Rocket Software'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;
}
```

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 requirements 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 the `<install_dir>\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, a snippet of which 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 the 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 a 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 the client on the server and also for the server's self authentication.

Look at the different properties files (server.properties, client.properties) and config files (`server.config` and `client.config`) in the `<install_dir>/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 the following properties in the server or client properties file:

Property	Description
`vbroker.security.disable=false`	The default value is `false`. If set to `true`, all security services will be disabled.
`vbroker.security.login=true`	If this property is set to `true`, during initialization, VisiSecure will attempt to log on to all realms listed by the property `vbroker.security.login.realms`.
`vbroker.security.authentication.config=cpp_server.config`	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`> as specified by `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 an SSL/TLS 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, see the descriptions for `vbroker.security.peerAuthenticationMode` in Security Properties for Java and Security Properties for C++.

Property	Description
`vbroker.security.login.realms=myrealm`	Provides 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`
	
	Note:
	`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.

Java Example: Authorization Using a Vault

To set up authorization for CORBA objects by using a vault, modify the following files in the `corbaauthz` example (located in the `<install_dir>\examples\vbroker\security` 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-and-paste Server.java)

- A newly-created vault named 'fault', as shown in the example below.

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 the `Bank.idl` file, make the following changes:

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

In the `AccountImpl.java` file, modify as indicated 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";            //  Add getRunAsRole()
    return null;                             //    to AccountImpl.java
  }                                          //
  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _converter.toSGD( _balance );
  }
  private float _balance;
  public static Bank.Converter _converter;
}
```

In the `server.java` file, to write the AccountManager’s reference to a file that can provide the client with 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();
}
```

Create new file `ConverterImpl.java`:

```java
// ConverterImpl.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 should be the same as the `server.java` file that is in the `corbaauthz` example. Just add the following line indicated by `ADD FOLLOWING LINE` comment:

```
// 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
      // *** UPDATE POA NAME TO "converter_poa" ***
      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"
          );

      // *** ADD FOLLOWING LINE ***
      ConverterImpl converterServant = new ConverterImpl(current);

      // Decide on the ID for the servant
      // *** CHANGE SERVANT ID ***
      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`

   You are logged into the realm `myrealm`

7. Enter:

   `> 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. Enter:

    `> quit`

    You are now generating a Vault to `fault`.

To launch run-as alias

Open three different command-line consoles (command prompts on Windows).

Start the osagent.

1. In console1, launch the client as follows:

   `vbj -DORBpropStorage=java_client.properties Client`

2. When prompted, enter `borland` / `borland.`
3. In console2, launch the server as follows:

   `vbj -DORBpropStorage=java_server.properties Server`

4. In console 3, launch the currency converter server as follows:

   `vbj -DORBpropStorage=java_server.properties ConverterServer`

5. Console 3 displays:

```
*** 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 console 1, launch the 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 specify `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.

Using a 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.

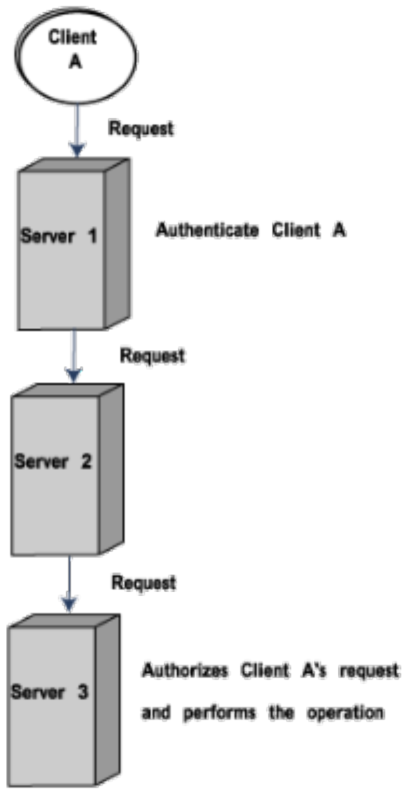
In a distributed environment, it is common for a mid-tier server to make *identity assertions* and act on behalf of the caller. The end-tier server must decide on whether the assertion is to be trusted or not. When propagating context, the client transfers the following information:

- **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.

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. See the following diagram:



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.

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

The inverse of impersonation, *delegation* is the form of identity assertion where the client explicitly delegates certain privileges to the server. In this case, the server is allowed to perform only certain actions as dictated by the client. VisiSecure performs only simple delegation.

Asserting identity of the caller

The identity assertion example illustrates the use of identity assertion APIs which can be used to explicitly assert an identity as caller. This example can be found at:

`<install_dir>\examples\vbroker\security\assertion`

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 suit your own environment before running the example. In this example, a server is setup with assertion trust and authorization rules.

To make the 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, to be a member of the Asserter role for the authorization domain `bank`.

Members of the role Asserter are described in the `bank.rolemap` as follows:

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

This means that 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, to be of Manager role for authorization domain "bank".

Members of the role Manager are 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 a 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.

The exception `CORBA::NO_PERMISSION` is thrown because 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` VisiSecure will attempt, at initialization time, to log on to all realms listed by property `vbroker.security.login.realms`.

Property	Description
`vbroker.security.authentication.config=cpp_server.config`	Specifies the path to the authentication configuration file.
`vbroker.security.peerAuthenticationMode=none`	This is to set the peer authentication mode. **none** - Authentication is not required. During SSL/TLS 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.<n>=Asserter@bank`	This property is used to specify a list of trusted roles (specify using 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 comma-separated list of available authorization domains.
`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>` as specified in `vbroker.security.authDomains`.

The following properties are set on the client side:

Property	Description
`vbroker.security.disable=false`	To enable the security service, set this property to `false`.

Property	Description
`vbroker.security.login=true`	If set to `true` VisiSecure will attempt, at initialization time, to log on to all realms listed by property `vbroker.security.login.realms`. On the client side, this property is set to `true`.
`vbroker.security.login.realms=myrealm`	
`vbroker.security.alwaysSecure=false`	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`	Sets the peer authentication mode. `none` - Authentication is not required. During SSL/TLS 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 which are trusted as mid-tiers, while others 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 Service 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. For more information, see sec-api-doc in the Help system, and the Security SPI for C++.

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.

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.

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 needs to assume an identity that has access to all resources that require the user to be in that role.

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/TLS).

VisiSecure functions in two transport environments:

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

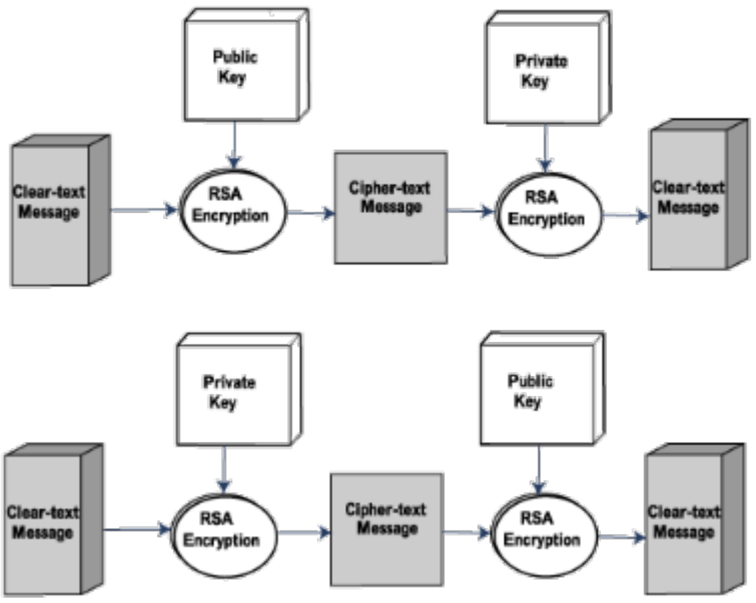
Encryption

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. Public/private key encryption is an example of *asymmetric* encryption. This is distinct from *symmetric* encryption, where the same *shared* key is used for both encrypting and decrypting the data.

Asymmetric encryption

Asymmetric encryption uses both a public and a private key. Both keys are linked 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, it 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.

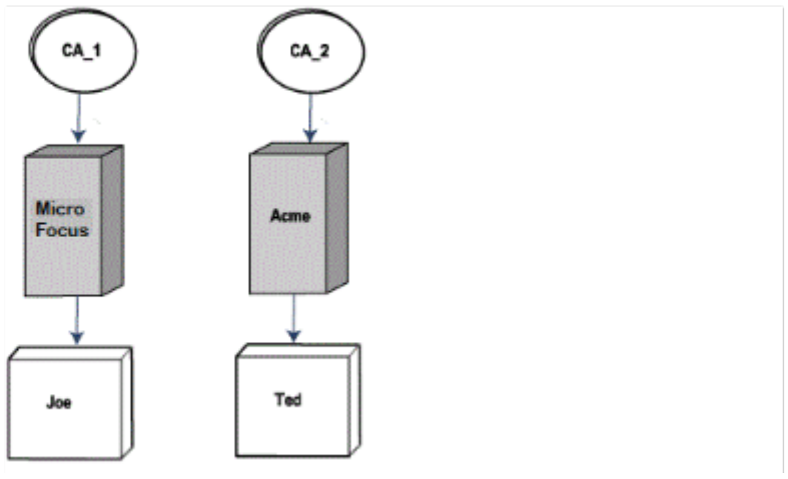
Distinguished names

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 user’s email address.	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 that certificate, `CA_2`.
3. Because `CA_2` is not a trusted CA, the server decides whether to accept or reject the connection.

Notes

- The manner in which you obtain certificate information from a CA is defined by that CA.
- A VisiBroker server using the MFCryptLib/OpenSSL security provider never trusts a peer certificate as a trustpoint (see "`vbroker.security.trustpointsRepository`" for details) if there is no valid CA intermediate/root certificate provided for it.

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 such as the `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 that 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.

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, thus implying that the document is unchanged and the signature was created by the owner of the public key.

Enabling SSL

For Java only:

VisiSecure for Java uses Java Secure Sockets Extension (JSSE) to perform SSL/TLS communication. The VisiSecure SPI Secure Socket Provider class provides access to the underlying SSL/TLS implementation. Any appropriate implementation conforming to the 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 other words, 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 the Java SDK is used.

Setting the level of encryption

The SSL/TLS 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 the cipher suites that is is prepared to use for the connection (this can be, if so configured, a subset of all the cipher suites that it knows). The server then takes this information and determines which of the client's requested cipher suites are supported by the server, and of those, which will be used for the connection. 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. However, you can modify the order of the cipher suites that are available and/or remove cipher suites you do not want to use.

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.

Note

There are important differences between the newer TLSv1.3 cipher suites and cipher suites from TLS versions up to TLSv1.2, still supported by VisiBroker. The new TLSv1.3 cipher suites cannot be used with TLSv1.2 and earlier protocol versions; and vice versa, cipher suites from TLS versions up to TLSv1.2 cannot be used with TLSv1.3.

The TLSv1.3 cipher suites use a different naming convention to those of the earlier TLS versions, in that they do not include the certificate type (e.g. RSA, ECDSA) or key exchange mechanism (e.g. DHE, ECDHE).

Segments of the name of the cipher indicate what the cipher suite does or does not provide. The general format for TLSv1.3 cipher suite names is:

TLS `_<Encryption Algorithm>_<Hashing Algorithm>`

For example:

- `TLS_AES_256_GCM_SHA384`

- `TLS_CHACHA20_POLY1305_SHA256`
- `TLS_AES_128_GCM_SHA256`

For older versions, up to and including TLSv1.2, the format is:

`[TLS|SSL]_<Key Exchange Algorithm>_<Authentication Algorithm>_WITH_<Encryption Algorithm>_<Hashing Algorithm>`

For example:

- `TLS_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA`

Unsupported cipher suites

Note that for security reasons:

- Cipher suites that include usage of the RC4 cipher are no longer supported by VisiBroker for C++.
- Cipher suites using Fixed Diffie-Hellman (that is, `TLS_DH_*` or `TLS_ECDH_*`) keys are no longer supported at all.
- Certificates signed with `MD5withRSA` are not supported at Security Level 1.

For further information

If you need to know more about cipher suite names or the possible values of the name segments, you can consult an SSL/TLS reference work.

Refer to OpenSSL for a list of supported ciphers at the current OpenSSL version. The specific version of OpenSSL included in your release is specified in its *Release Notes*.

Alternatively, the `openssl` utility included with the VisiBroker distribution can be used to report which cipher suites are supported bu VisiSecure for C++:

`<install_dir>/bin/openssl -v -s -stdname ALL`

The list of supported cipher suites for VisiSecure is determined by the underlying security implementation. For VisiSecure for Java, this is the JSSE package used, which in turn depends on the version and manufacturer of the JDK used. For VisiSecure for C++, you can find the list of supported ciphers by using the ssl::CipherSuiteInfoList* listAvailableCipherSuites() API.

Using ECDHE cipher suites

VisiBroker supports the use of ECDHE cipher suites. To enable these cipher suites a private key and certificate chain must be provided that conform to the following requirements. For ECDHE, certificates must contain an identity key from one of the supported DH Groups (check the Release Notes). The supported cipher suites are:

TLS v1.0:

- `TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA`
- `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA`
- `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA`

TLS v1.2:

- `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`

For ECDHE using RSA certificates, the server's private key which matches the public key in the server's identity certificate must be an RSA key that is authorized for signing. The supported cipher suites are:

TLS v1.0:

- `TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`

TLS v1.2:

- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

Notes

- When using TLSv1.3, the displayed cipher suites will not contain 'ECDHE_' in their titles. However, the same technology will be used to exchange ephemeral keys and provide the same Perfect Forward Secrecy.
- The ECDH_ECDSA and ECDH_RSA groups of cipher suites are not supported in this release.
- In addition, in all cases the following requirements must be met:
  - The identity certificate must be signed by a capable ECDSA or RSA (as appropriate) certificate (intermediate or root).
  - The complete certificate chain must be provided.
  - For a server, the ECC private key that corresponds to the identity certificate must be provided.

ECC Curves

OpenSSL supports many pre-defined ECC curves (also known as 'named curves' or ‘elliptic curves’) that can be used by ECDHE ciphers. A complete list of the supported named curves can be obtained by executing the following command, using the `openssl` utility:

```
<install_dir>/bin/openssl ecparam -list_curves
```

VisiBroker for C++ supports the well-known TLS elliptic curves as defined in the following IANA RFC:

- RFC4492: http://www.iana.org/go/rfc4492
- RFC7027: http://www.iana.org/go/rfc7027

You are free to select any of the curves from the supported list (provided the key length is a minimum of 256 bits; see Note below) when generating your ECC keys/certificates. OpenSSL treats keys generated using these curves transparently.

While you may use any of the supported curves, the following curves were selected for the purposes of verifying the behavior of VisiBroker 8.5:

- `prime256v1`
- `secp384r1`

Note

VisiBroker for C++ inherits OpenSSL's default behavior, which is to disable all elliptic curves weaker than 256-bit unless configured otherwise. VisiBroker for Java relies on the underlying JSSE implementation for working with ECC curves.

ECDHE ephemeral keys

In previous releases, elliptic curve key groups were used to identify and sign entities and certificates. Elliptic curve key groups form a subset of Diffie-Hellman key groups. Starting with VisiBroker 8.5.7, the use of elliptic curve definitions were deprecated. These have been replaced by the use of Diffie-Hellman key groups (DH Groups). The remaining supported safe elliptic curve definitions are a subset of our supported DH Groups.

When a DH Group identity is specified as part of the server's certificate and key configuration, ECDHE ephemeral keys are created using the same DH Group that is chosen for the server's primary key.

If no DH Group identity is detected, the ECDHE cipher suites require that a DH Group is chosen to create the ECDHE temporary keys that will be used in the session. The DH Group will be selected automatically. However, you can also configure a prioritized set of DH Groups that will be used in the TLS handshake to negotiate a DH Group to be used with the peer. See the options vbroker.security.server.socket.TLSCipherGroups and vbroker.security.client.socket.TLSCipherGroups.

The vbroker.security.server. socket.ecdheCurve property is now deprecated. It is, however, still supported in this release in order to maintain existing configurations. Its value is treated in the same way as vbroker.security.server.socket.TLSCipherGroups.

Note

If `prime256v1` is specified, `secp256r1` will appear in the logs. This is because a previous ANSI X9.62 standard (Public Key Cryptography For The Financial Services Industry) defined some of the same curves as the IANA list, but with different names. Where this occurs either name can be used in the VisiBroker configuration. For example, the IANA `secp256r1` is the same as the ANSI `prime256v1`.

Enabling Security

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

`vbroker.security.disable=false`

Enabling SSL/TLS

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

To disable SSL/TLS

To disable SSL/TLS on the client side, set the following property as shown:

`vbroker.security.secureTransport=false`

To disable SSL/TLS on the server side, use the following property setting:

`vbroker.security.server.transport=CLEAR_ONLY`

Setting the Log Level

Activity logging in VisiBroker employs one or more Logger objects. Using the Default Logger results in security log messages being interleaved with the main ORB's logging output. If it is required that the security logging output be isolated, one or more additional loggers can be created.

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 output to various independent end points.

The 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*.

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

VisiSecure for Java

Messages from VisiSecure for Java internal are logged under the source name "secure".

To enable VisiSecure for Java to log messages at level `info` and below, set the following properties:

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

The default value of `vbroker.log.default.filter.secure.logLevel` is `debug`.

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

VisiSecure for 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 an appropriate destination or destinations. The C++ ORB and its services use this mechanism for the output of any error, warning or informational messages.

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

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 of `vbroker.log.default.filter.v_seccsiv2.logLevel` is `debug`.

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

Types of message	Source name
Authentication-related messages	`v_secauthn`
Authorization-related messages	`v_secauthz`
SSL-related messages	`v_secssl`

Types of message	Source name
CSIv2-related messages	`v_seccsiv2`

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 (neither classes nor 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.

Browser considerations

Some browser versions require the installation of the CA certificate before allowing an IIOP/HTTPS connection. Follow these guidelines to use IIOP/HTTPS with such browsers:

- Make sure your server certificates are issued by a CA that is already trusted.
- Install the root certificate 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 invalid site certificates** is not checked to use a server certificate that does not contain the host name of the server.

Quality of Protection

VisiBroker's extensions of standard CORBA policies include an implementation of Quality of Protection (QoP) which provides another level of fine-grained control over your run-time security requirements.

There are two types of QoP:

- Server QoP is usually installed on a Portable Object Adapter (POA).
- Client QoP is usually installed on a object.

Setting properties and QoP

There are several properties that can be used to ensure the Quality of Protection of a connection. These properties can be used to fine-tune connection quality.

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

- `vbroker.security.TLS13CipherSuites` for TLSv1.3.
- `vbroker.security.cipherList` for TLSv1.2 and earlier.

These properties 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.

QoP properties can also be set programmatically for servers and clients, using `ServerQoPConfig` and `ClientQoPConfig` respectively. For more information, see Configuring Quality of Protection(QoP).

These APIs allow you to set target trust (whether or not targets must authenticate), and 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)

Configuring QoP for the 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 );
  }
}
```

The `disable` parameter determines whether security is disabled/enabled for this POA. When security is disabled, the remaining settings become irelevant. When enable, the allowable values for the remaining parameters are described below.

- `transport` has three possible values:
  - `CLEAR_ONLY` : Uses only clear listener to accept request.
  - `SECURE_ONLY` : Uses only SSL listener to accept request.
  - `ALL` : Uses both clear and SSL.
- `idType` has the possible values of `com.borland.security.csiv2.ServerQoPPolicy` .

Value	Description
`NO_ID`	Expecting no identity
`UP`	Expecting Username Password identity

Value	Description
`PK`	Expecting transport identity
`UP_AND_PK`	Expecting both to be present and valid
`UP_OR_PK`	Expecting either one to be present and valid

- `enableIdAssertion` determines whether this server can (`true`) / cannot (`false`) accept caller identity propagated through a 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` is, for authorization purposes, the AccessPolicyManager responsible for this POA. The default value is 'null' meaning there is no authorization.

To configure QoP for the server, follow the steps as given below:

1. To enable access controls, set `disable` to `false`
2. For a method to be secure, set `transport` to `SECURE_ONLY`
3. For the server to require a client's credentials for authentication, set `trust_in_client` to `true`.

   For more information on transport methods and other QoP related parameters, see class vbsec::ServerConfigImpl.
4. To create server QoP configuration object

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

5. 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 the client

The initial step of creating a QoP is to create a `QoPConfig` and use it to specify the security requirements that are to be enforced.

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 configuration; will throw `PolicyError`.	For outgoing, use clear IIOP transport.
`SECURE_ONLY`	For outgoing, use SSL and make sure that server certificate is trusted.	For outgoing, use SSL and server certificate can be non-trusted.
`USE_ANY`	Invalid configuration; will throw PolicyError.	For outgoing, try SSL then fallback to clear when fail, server certificate 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 the client, QoP is set as follows:

For method to be secure, set `transport` to `SECURE_ONLY` and set `trustInTarget` to `false`. (With these settings, the 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 an `Any` object:

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

3. Narrow the reference to `Bank.AccountManager` 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:

```
...
  try {
    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 some 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:

C++

See QoP API.

Java

See `com.borland.security.csiv2` and Security Properties for Java.

Creating custom plugins

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

- LoginModules
- CallbackHandlers
- Authorization Service Providers via the SPI
- Trust Providers via the SPI

LoginModules

`LoginModule` describes the interface implemented by authentication technology providers. Login modules are plugged in beneath applications to provide a particular type of authentication.

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

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

`LoginModule` serves as the parent of all login modules. User plug-in 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 for authenticating the given Subject and associating relevant Principals and Credentials with the subject. They are also responsible for removing and disposing of such security information during logout.

To use the `LoginModule` it must be included in the authentication configuration file alongside any other required login modules. At run-time, the configured login modules will 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 (for example, to retrieve a user name and password). If so, you will need to become familiar with the readily available `CallbackHandler` interfaces. (Alternatively, you can create your own Callback implementations.)

The LoginModule will invoke the `CallbackHandler` as passed to the LoginModule's `initialize` method.

If a login module's implementation has no end-user interactions, it would not require the use of the callback feature.

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 login module 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.

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

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

For more information on implementing login modules, see the *Login Module Developer’s Guide* in your JDK's JAAS documentation.

In addition to these methods, a `LoginModule` implementation should provide a public constructor with no arguments. This allows for its proper instantiation by a `LoginContext`. 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 login module 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 succeeds (the relevant `REQUIRED`, `REQUISITE`, `SUFFICIENT`, and `OPTIONAL LoginModules` succeeds), 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 fails (the relevant `REQUIRED`, `REQUISITE`, `SUFFICIENT` and `OPTIONAL LoginModules` does 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 callback handlers provided, or your own custom callback handler. For information about this property, refer to Security Properties for C++ or Security Properties for Java as appropriate.

See VisiSecure for C++ APIs for more details.

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

Implementations of the callback interface are passed to a CallbackHandler, allowing underlying security services that have 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 to the underlying security services.

Authorization Service Providers

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 that are granted access to certain resources. Whenever an access decision is going to be made, the `AuthorizationServiceProvider` is consulted. The Authorization Service is closely associated with the Authorization domain concept. One Authorization Service is installed for each Authorization domain implementation, and functions only for that particular Authorization domain.

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

Use the following property to set the implementing class for the `AuthorizationServiceProvider`:

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

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

Another important feature 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 AuthorizationServiceProvider is again 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 the domain itself.

To plugin an authorization service, you need to set these 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 rolemap mechanism.

`USE_STD_NS` is a definition set up 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 the name of the module:

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

To print out the options given to the custom authorization 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 members of the "Manager" role, and for the `Account` object clients should be members of either the "Customer" or "Teller" roles:

1. To create the 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));
```

2. To create the 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));
```

3. To create the 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;
}
```

4. Implementation of the functions of the `CustomPermissionCollectionImpl` 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 );
...
```

5. 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;
}
```

6. To check if one of the principals matches the 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;
  }
}
```

7. To check if at least one 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 the JAAS configuration file. The example client and server uses username/password authentication of the client on the server and also for the server's self-authentication.

Look at the different properties files (`server.properties` , `client.properties`) and config files (`server.config` and `client.config`) in the `<install_dir>/examples/vbroker/security/corbaauthz` folder.

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

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

To enable security, you must set up the following properties in the server or client properties file:

Property	Description
`vbroker.security.disable=false`	The default value is `false` . If set to `true` , all security services are disabled.
`vbroker.security.login=true`	If this property is set to `true` , at initialization time the ORB will attempt to log on to all the realms defined by the property `vbroker.security.login.realms` .
`vbroker.security.login.realms=myrealm` `vbroker.security.login.realms=GSSUP#myrealm`	Defines the list (comma-separated) of realms to login to. This is used when login takes place, either through property `vbroker.security.login` (set to `true`) or API login.

Property	Description
`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, see the property descriptions provided by Security Properties for Java or Security Properties for C++.
`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.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`

Property	Description
`vbroker.security.logLevel=LEVEL_DEBUG`	Use this property to control the degree of logging.

Trust Providers

You can also plug in the assertion trust mechanism. Assertion can happen in a multi-hop scenario, or can be explicitly called through the 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 settings to configure trusted peers on the server side. At 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 plug in 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 plug-in following the same mechanism as above.

Creating Secure CORBA Applications Using Java

This section describes the use of VisiSecure to make secure connections for CORBA applications using Java.

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. Wherever appropriate the usage models for clients and servers are discussed separately. 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

An identity can be a username/password/realm triad, or certificates can be used. These can be collected through JAAS modules or through APIs.

Clients

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

cServers

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.

Keeping these constraints in mind, the VisiSecure Server supports the following usage models, any of which could be used 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 via property settings
- Certificate-based authentication, using KeyStores via APIs
- Certificate-based authentication, using APIs
- pkcs12-based authentication, using KeyStores
- pkcs12-based authentication, using APIs

Username/password authentication, using JAASmodules, 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 via 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 via APIs

You can use the same APIs discussed in Username/password authentication, using APIs to login 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 `CertificateWallet` 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 Quality of Protection. 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/TLS connections to set 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. For QoP API information, go to the VisiSecure for Java API and SPI book.

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 set the `javax.net.ssl.trustStore` property to point to that. 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.

1. Construct a `SecureRandom` object and seed it.
2. 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, see the *VisiSecure for Java API and SPI*.

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 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*.

Examining SSL related information

VisiSecure provides APIs to inspect and set SSL-related information. The `SecureContext` API is used to specify a Trust Manager, PRNG, inspect the SSL/TLS cipher suites, and enable ciphers.

Clients

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

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

The Bank SSL example included in the `<install_dir>/examples` 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/TLS 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:

1. Ensure that the VisiBroker Smart Agent (osagent) is running on your network.
2. Build the example in the directory by typing:

   `make -f Makefile_java` on UNIX

   or:

   `nmake /f Makefile_java` on Windows

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

3. The following steps are the recommended way to add certificate-chain identity, to construct a `CertificateWallet`, and to log in with a security context.

   The old method of inserting a certificate chain by using API `setPKprincipal` using `(byte [] []derCertChain, byte[] privateKey, String passPhrase))` and resolving the initial reference of `SecurityCurrent` on the ORB, still exists for backward compatibility. In this case, the requirement is the `setPKPrincipal` API should be called prior to calling `resolve_initial_references` API.

4. To make the server run in the background, enter the following command:

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

   or on Windows:

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

5. To make 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 &`

   or on Windows:

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

6. To connect using the Java SecureClient:

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

7. To connect using the C++ SecureClient:

   `prompt>SecureClient -DORBpropStorage=cpp_client.properties`

8. To set up the 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 ()
};
```

9. 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());
```

Creating Secure CORBA Applications Using C++

This section describes the use of VisiSecure to make secure connections for CORBA applications using C++.

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 appropriate, the usage models for clients and servers are discussed separately. 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

An identity can be a username/password/realm triad, or certificates can be used. These can be collected through LoginModules or through APIs.

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

Clients

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

Servers

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 the certificate chain.

Keeping these constraints in mind, VisiBroker supports the following usage models, any of which could be used to provide an identity to the server or client:

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, 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 via 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 via APIs

You can use the same APIs discussed in Username/password authentication, using APIs to login 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 `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 `CertificateWallet` 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 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/TLS connections to set 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 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 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.

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 need 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/TLS server connection manager (scm).
- `<IIOP_SCM_NAME>` is the name of an IIOP scm defined within the same server 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. The default value is `ThreadPool`. (For more information, see the *VisiBroker for C++ Developer's Guide*.)

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
```

Examining SSL related information

VisiBroker provides APIs to inspect and set SSL-related information. The `SecureContext` API is used to inspect the SSL/TLS cipher suites and enable selected ciphers.

To examine the peer certificates, whether on the client or server side, use `getPeerSession` of the `vbsec::Current` object to obtain the `SSLSession` object associated with the target (if client) or thread (if server). You can then use `getPeerCertificates()` to access the `CORBAsec::X509CertList`.

SSL example

This section demonstrates how to make a minimal SSL/TLS configuration for the client and server to communicate using SSL/TLS 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 code 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=vbsec`

4. Launch the client using the command below:

   `prompt> ./Client -DORBpropStorage=cpp_client.properties -Dvbroker.orb.dynamicLibs=vbsec`

5. Open the property files `cpp_server.properties`, `cpp_client.properties` and notice how certificates and private keys are installed using the wallet property set in each file.
6. Browse through the content of subdirectory identities and trustpoints and understand how the directory wallet and trustpoints are structured.

The value of `vbroker.orb.dynamicLibs` specifies the security library name. You can either:

- Enter `vbsec`, as in the example above. This loads VisiSecure (on any platform) without your needing to know the full pathname, but only works if the VisiBroker `lib` directory is present on the shared library load path.
- Enter the full shared library name and path, including the prefix and shared library suffix. Note that the VisiSecure shared library name depends on the operating system and platform. 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 if you intend to use this option.

Using initializers to install certificates, private key, trustpoints and CRL

This section is similar to the 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`,
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 in 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 the additional property `-Dvbroker.app.useCRL` and therefore launching the server, for example, becomes:

   ```
   prompt> ./Server -DORBpropStorage=cpp_server.properties -Dvbroker.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 applications that are written with VisiSecure in mind can take full advantage of the control and power of VisiSecure features using APIs.

1. Build this example in this directory by executing the command:

   `nmake cpp` (for Windows) or `make cpp` (for UNIX)

   When the build succeeds, the executables

   - `SecureServer.exe` / `SecureClient.exe` are created on Windows
   - `SecureServer` / `SecureClient` are created on Linux/UNIX

2. Make sure the osagent is up and running.
3. Launch the server using the command below:

   `prompt> ./SecureServe`

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/TLS 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 PKCS#12 storage, a 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 the build succeeds, the executable `pkcs12Server.exe` is created on Windows.

2. Make sure the osagent is up and running.
3. Launch the server using the command below:

   `prompt> ./pkcs12Server frans.pfx frans`

4. Launch the client using the command below:

   `prompt> .SecureClient`

5. 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`

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

Secure Object URLs

Section *7.6.10 Object URLs* of the CORBA specification defines two URL schemes for CORBA Objects: `corbaloc` and `corbaname`. As of VisiBroker 8.5 SP8, the ORB will accept URLs that describe CORBA servers that can support Transport Level Security (TLS) connections. The format of such URLs are described in the following sections.

These Object URLs are often used to specify bootstrap values of the ORB service initial references, however, they can be used more broadly in any circumstance where a stringified IOR is required, for e.g. as an argument to:

`ORB->string_to_object(...);`

corbaloc:ssliop Object URLs

An example format of a typical corbaloc URL for TCP (or IP) based transport protocol is:

`corbaloc : [<protocol>] : [<version> @] <host> : [<port>]  <stringified_object_key>`

The defaults (if not specified) for are:

Field	Value
protocol	`iiop`
GIOP version	`1.0`
IIOP port	`2809`

An unsecured CORBA service listening only for the IIOP protocol might therefore be reachable with URLs such as:

```
corbaloc:iiop:1.0@localhost:2809/SomeService
corbaloc::localhost:/SomeService
```

The above URLs are equivalent to each other.

VisiSecure additionally supports the protocol `ssliop` which can be used to tell a client that the CORBA server, which it should be connected to, is listening to accept GIOP over SSL/TLS on a specified host and port combination. Below are some examples of `ssliop` Object URLs:

```
corbaloc:ssliop:1.2@127.0.0.1:23232/NameService
corbaloc:ssliop:myhost.foo.com:54321/
%00PMC%00%00%00%04%00%00%00%06/test%00%20%20%00%00%00%06server
```

To interoperate with other ORBs, alternate values of `<protocol>` that are allowed and interpreted as equivalent to `ssliop` are:

- `tlsiop`
- `iiops`
- `it_iiops`
- `fssliop`

There is no default value for `<port>`. The default protocol for `<version>`, if none is specified, is `1.0`, just as is the case with the protocol `iiop`.

URLs for servers with more than one listening endpoint (secure or unsecured) can be specified in a single URL:

```
corbaloc:ssliop:1.2@myhost.foo.com:20003,
    iiop:1.2@myhost.foo.com:20002/NameService
```

The above example denotes a CORBA Naming Service running on a host named `myhost.foo.com`, listening for GIOP 1.2, configured for secure traffic on port 20003 and an insecure traffic on port 20002. Any length sequence of IOR profile protocol with address values can be listed delimited by `,` in this manner.

corbaname:ssliop Object URLs

The `corbaname` general format of an Object URL is:

`corbaname: [<protocol>] : [<version> @] <host> : [<port>] / [<nameserver_key>] [# <naming_ctx>]`

An unsecured example might be:

```
corbaname::127.0.0.1:20002/NameService#MyCtx/MyServerName
```

These Object URLs can also support the `<protocol>` value ssliop. Usage and defaults are identical to those for corbaloc. If the NameServer in the above example also listened on port 20003 for secure connections, then a URL to access it securely may be written:

```
corbaname:ssliop:127.0.0.1:20003/NameService#MyCtx/MyServerName
```

Consistent with the corbaloc multiple protocol with address values, these can be specified delimited by `,`.

Quality of Protection for Secure Object URLs

Secure IORs typically contain information to tell clients what level of TLS authentication is possible and/or required from them for a successful connection establishment to the server listening endpoint. These values are known as `AssociationOptions`. Two pieces of such information are encoded within the `TAG_TLS_SEC_TRANS` tagged component within a normal IOR. One describes the capabilities of the target server, the other what the target server will require of the connecting client. Section *10.5.1.3 TAG_TLS_SEC_TRANS* of the CORBA specification describes the semantics of the various possible Association Options.

The `corbaloc` and `corbaname` URL formats do not define any way to encode this information for the clients that read them. Instead, VisiBroker provides two configuration properties that allow values to be specified for the absent `AssociationOptions`. All Object URLs read by the ORB instance will be interpreted to have those values in their `TAG_TLS_SEC_TRANS` component. The property names and their default values if not set are:

```
vbroker.security.corbaloc.target_supports=EstablishTrustInTarget
vbroker.security.corbaloc.target_requires=0
```

This default tells the client that the server's identity can be verified, if the client choses to do so, from the certificate that will be received during the initial handshake.

The full or shorthand names for the Association Options in the table below can be combined with a pipe character (no spaces are allowed), or the decimal values can be combined as the sum of the required options.

Association Option	Shorthand	Decimal Value
Integrity	INT	2
Confidentiality	CF	4
EstablishTrustInTarget	ETT	32
EstablishTrustInClient	ETC	64

For example, the following pairs of properties are all equivalent and specify that full TLS mutual authentication of both client and server should occur:

```
vbroker.security.corbaloc.target_supports=
    EstablishTrustInTarget|EstablishTrustInClient
vbroker.security.corbaloc.target_requires=EstablishTrustInClient
vbroker.security.corbaloc.target_supports=ETT|ETC
vbroker.security.corbaloc.target_requires=ETC
vbroker.security.corbaloc.target_supports=96
vbroker.security.corbaloc.target_requires=64
```

Security Properties for Java

Property	Description	Default
`vbroker.security.logLevel`	Use this property 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 or not the transport connection is encrypted. 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 set to `true` then transport QoP is set to `SECURE_ONLY`, which means the client will only accept secure transport. If either of them is set to `false` then Client does not mandate security at the transport layer.	`false`

Property	Description	Default
`vbroker.security.server.transport`	This property is used on the server side to define 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 property `secureTransport` is `true`.	`SECURE_ONLY`
`vbroker.security.server.requireUPIdentity`	Set this to `true` if the server requires the client to send a Username/Password for authentication (regardless of certificate-based authentication). This is a server-side property.	`false`
`vbroker.security.disable`	If set to `true`, disables all security services.	`true`

Property	Description	Default
`vbroker.security.transport.protocol`	This property is used to select a security transport protocol. For a standard list of protocol version names, go to https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html By default, VisiBroker for Java establishes the highest TLS version supported by the underlying Java VM JSSE provider, and uses that as the preferred maximum TLS version. For information on these protocols, see the Oracle Corporation documentation.	Highest TLS version supported
`vbroker.security.requireAuthentication`	Server-side only property used to specify whether the client is required to authenticate.	`false`

Property	Description	Default
`vbroker.security.enableAuthentication`	**Note:** This property is deprecated. See Authentication for recommended methods of specifying authentication. Server-side only property. This back-compatible property is used for supporting `PasswordBackEnd` style authentication. When set to `true`, the program will try to construct the specified `PasswordBackEnd` for authenticating.	`false`

Property	Description	Default

`vbroker.security.authentication.callbackHandler`	Specifies the callback handler used for login modules to use for interacting with user 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

`vbroker.security.authentication.config`	This specifies the path to the configuration file used for authentication.	null
`vbroker.security.authentication.retryCount`	Number of times to retry if remote authentication failed.	`3`

Property	Description	Default
`vbroker.security.authentication.clearCredentialsOnFailure`	By default, if the authorization realm finds the authenticator is incorrect after the maximum number of retries have been attained, 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`, during initialization the ORB will attempt to log in to all the realms listed by property `vbroker.security.login.realms`.	`false`
`vbroker.security.login.realms`	This gives a list of comma-separated realms to log into. This is used when login takes place, either through property `vbroker.security.login` (set to `true`) or API login using `login()`.	n/a

Property	Description	Default
`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 to reacquire authentication information using the `CallbackHandler`. This property requires the callback handler to be set either using the appropriate property or at runtime by calling the appropriate method.	`false`

Property	Description	Default
`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. 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`	Specifies a comma-separated list of available authorization domains. For example: `vbroker.security.authDomains=<dom1>,<dom2>…`	null

Property	Description	Default
`vbroker.security.domain.<domain_name>.rolemap_path`	Specifies the location of the Role DB 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`, enables dynamic loading of the Role DB file as specified by the `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`	Specifies the rolemap refresh time in seconds.	`300`

Property	Description	Default
`vbroker.security.domain.<domain name>.runas.<run_as_role_name>`	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
`vbroker.security.domain.<domain_name>.defaultAccessRule`	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.security.peerAuthenticationMode`	Sets the peer authentication mode. 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`

Property	Description	Default
`vbroker.security.trustpointsRepository`	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 set to `true`, the JSSE default trust files like `cacerts` and `jssecacerts`, if present in JRE, will be used to load trusted certificates.	`false`

Property	Description	Default
`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 assertion made by the `ServerAdmin` role in the `default` authorization domain.	n/a
`vbroker.security.assertions.trust.all`	Setting to `true` will trust all the assertion made by peers.	`false`

Property	Description	Default
`vbroker.security.cipherList`	Set this property to a comma-separated list of 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?/TLS cipher suites. If this property is set but no certificates are configured, all non-anonymous cipher suites specified as part of this property's value are ignored; only the anonymous cipher suites specified as part of this property will remain actively available for the SSL handshake.	n/a
`vbroker.security.controlAdminAccess`	Set this to `true` for enabling Server Manager operations on a Secure Server.	`false`

Property	Description	Default
`vbroker.security.serverManager.authDomain`	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`	Specifies the role name required for accessing all Server Manager operations.	n/a
`vbroker.security.serverManager.role.<method_name>`	Specifies the role name required for accessing the specified method of the Server Manager.	n/a

Property	Description	Default

`vbroker.security.wallet.type`	A wallet is a set of directories containing encrypted private keys and certificate chains for each identity. The possible values are:
	
	`Directory:<path_to_identities>`
	
	Use the `Directory` value to point to the directory containing the directories for all identities.
	
	`PKCS12:.<path_to_PKCS#12_KeyStore>`
	
	Use the `PKCS12` value to configure the `PKCS#12` keystore directory.
	See PKCS#12-based authentication using KeyStores for details.
		n/a

Property	Description	Default
`vbroker.security.wallet.identity`	If the `vbroker.security.wallet.type` is set to `Directory`, use this property to point to a sub-directory within the path defined by `vbroker.security.wallet.type` that contains keys and/or certificate information for a specific identity. Note that the value of this property must consist only of lower-case letters. If `vbroker.security.wallet.type` is set to `PKCS12`, the VisiBroker for Java secure client then looks for a file `<identity>.p12` in the `<path_to_PKCS#12_KeyStore>` folder.	n/a
`vbroker.security.wallet.password`	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 limit before re-authentication is required.	600 sec

Property	Description	Default
`vbroker.security.TSS.stateful`	This corresponds to the `CompoundSecMechList` IOR component's 'stateful' field (see OMG specs). This signifies whether or not the server supports 'stateful' Security Attribute Service (SAS) sessions and therefore the client can make decisions according to the standard behavior as defined by the OMG specifications.	`true`
vbroker.security.TSS.sweepThreshold		
vbroker.security.TSS.sweepPeriod		
vbroker.security.TSS.invokeCallbackOnTTL		

Property	Description	Default

`vbroker.security.trustProvider`	This property is a part of the trust provider plugability mechanism:
	
	`vbroker.security.trustProvider=xyz` where `xyz` can be any string.
	
	The FQCN (fully qualified class name) 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

Property	Description	Default
`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`, the corresponding bit will be set in the SASContextSec component. When set to `false`, it will be reset. This bit signifies whether or not the server supports identity assertion and therefore, the client can react according to the pre-defined behavior associated with this bit. (See OMG specifications.)	`true`
`vbroker.security.client.supportNoDelegation`	If set to `true`, the client will add support for NoDelegate in `TAG_SSL_SEC_TRANS` tag.	`false`

Property	Description	Default

`vbroker.security.server.socket.enabledProtocols`
 	
`vbroker.security.client.socket.enabledProtocols`
	These properties specify the SSL/TLS protocol version, such as TLSv1.2 and TLSv1.3, during the SSL/TLS handshaking process when using the VBJ security module, on the server and client sides respectively. The values should be a list of comma-separated values indicating the enabled protocols.
	
	(**Note:** The possible values are different for each underlying JSSE implementation. Please refer to the corresponding JSSE reference guide for the available values.)
	
	The default depends on the JDK vendor. Note that the highest available protocol version is determined by the installed JSSE implementation.
		Depends on the JDK vendor.

Property	Description	Default
`vbroker.security.server.ssl.handshakeTimeout`	This property specifies the maximum time (in milliseconds) for the SSL/TLS handshake to complete at the server side. It can help to prevent the server from hanging due to unresponsive clients during SSL/TLS handshake. The default timeout is 5000ms. To disable the timeout, set it to 0.	5000
`vbroker.se.iiop_tp.scm.ssl.listener.trustInClient`	A server-side property. Set 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.se.iiop_tp.scm.ssl.manager.type`	Specifies the type of Server Connection Manager. The possible values are `Socket` and `Socket_nio`.	`Socket`

Property	Description	Default
`vbroker.se.iiop_tp.scm.ssl.listener.selectorMax`	Maximum number of NIO Selectors that can be created in the NIO Selector Pool. Increasing this value may improve the throughput in cases of very high concurrent invocations. This property is only applicable if `Socket_nio` is configured for `vbroker.se.iiop_tp.scm.ssl.manager.type`.	20

Property	Description	Default
`vbroker.security.CRLRepository`	Specifies the directory where you want the list of serial numbers of revoked certificates (Certificate Revocation List (CRL)), issued by the Certificate Authority (CA), to reside. All files in the directory will be loaded and included in the CRL — and thus interpreted as no longer valid. CRL files must be in DER format. Once the CRLs are loaded, VisiSecure examines all certificates sent by a peer during SSL/TLS handshake. If any of the peer certificates appears in the CRLs, an exception will be thrown and the connection will be refused. For more information, see Certificate Revocation List (CRL) and revoked certificate serial numbers.	n/a

Property	Description	Default
`vbroker.security.CSS.strict`	This property is used to configure the behavior of the Client Security Service (CSS) when an exception is received and the Security Attribute Service (SAS) Context is not set by the Target Security Service (TSS). If this property is set to `false` (default), the CSS will simply propagate the exception received. If this property is set to `true`, the CSS throws a `BAD_PARAM` exception instead, stating that the SAS Context is missing.	`false`

`vbroker.security.CSS.throw_ssl_exceptions`	Determines the form of exception messages thrown by the CSS in response to SSL/TLS connection errors. An example of such as connection error might be when a client tries to make a fresh request over an SSL/TLS connection that has been very recently closed by the connection idle and closure mechanism.
	
	If this property is set to `false` (default) then a `CORBA_BAD_PARAM` error is thrown, such as:
	
	`org.omg.CORBA.BAD_PARAM: CSIV2 Protocol error: TSS did not respond with a SAS context vmcid: 0x0  minor code: 0  completed: No`
	
	If this property is set to `true` then the exception thrown instead is a `NO_PERMISSION`, as is usual for SSL/TLS exceptions in VisiBroker. For example:
	
	`org.omg.CORBA.NO_`
		`false`

Property	Description	Default
	`PERMISSION: SSLException: javax.net.ssl.SSLException: Connection has been shutdown: javax.net.ssl.SSLException: java.net.SocketException: Broken pipe  vmcid: 0x0 minor code: 0 completed: No`	
`vbroker.security.keyStore.keyPass`	This property is used to specify the keypass value of the Java Keystore. If not set, the value set by the JVM vendor's `javax.net.ssl.keyStorePassword` property will be used for the keypass.	n/a
vbroker.security.support.gatekeeper.end2endCSIV2		
vbroker.security.support.gatekeeper.replyForSAS		

Property	Description	Default
vbroker.security.authenticateDelegatedUPIdentity		

SSL/TLS Server Connection Manager properties

The following table lists the SSL/TLS Server Connection Manager (SCM) properties.

In this table, possible values for `<se_name>` are:

- `iiop_tp`
- `iiop_ts`

Property	Default	Description
`vbroker.se.<se_name>.scm.ssl.manager.type`	`Socket`	Type of the Server Connection Manager. Possible values are `Socket` and `Socket_nio`.
`vbroker.se.<se_name>.scm.ssl.manager.connectionMax`	`0`	The maximum number of cached connections on the server. The default `0` means that there is no restriction.
`vbroker.se.<se_name>.scm.ssl.manager.connectionMaxIdle`	`0`	Specifies the time, in seconds, which the server uses to determine if an inactive connection should be closed. If a cached connection has been idle longer than this time, then the server closes the connection.
`vbroker.se.<se_name>.scm.ssl.listener.type`	`SSL`	The type of protocol that the listener is using.

Property	Default	Description
`vbroker.se.<se_name>.scm.ssl.listener.port`	`0`	Specifies the port number to be used with the host name property. `0` means that the system will pick a random port number.
`vbroker.se.<se_name>.scm.ssl.listener.proxyPort`	`0`	Specifies the proxy port number to be used with the proxy host name property. 0 means that the system will pick a random port number.
`vbroker.se.<se_name>.scm.ssl.dispatcher.type`	`ThreadPool`	The type of thread dispatcher used in the Server Connection Manager. Possible values are `ThreadPool` and `ThreadSession`.
`vbroker.se.<se_name>.scm.ssl.dispatcher.threadMin`	`0`	Specifies the minimum number of threads that the Server Connection Manager must create.
`vbroker.se.<se_name>.scm.ssl.dispatcher.threadMax`	`0`	Specifies the maximum number of threads that the Server Connection Manager can create.
`vbroker.se.<se_name>.scm.ssl.dispatcher.threadMaxIdle`	`0`	Specifies the time, in seconds, before an idle thread is removed from the thread pool. Note that using the value `0` (default) means that idle thread harvesting will **not** be carried out.

Property	Default	Description
`vbroker.se.<se_name>.scm.ssl.connection.tcpNoDelay`	`false`	Specifies whether `tcp_nodelay` should be set on the socket.
`vbroker.se.<se_name>.scm.ssl.listener.selectorMax`	`20`	Specifies the maximum number of NIO Selectors that can be created in the NIO Selector Pool. Increasing this value may improve the throughput if there is a very high number of concurrent invocations. This property is only applicable if `Socket_nio` is configured for `vbroker.se.<se_name>.scm.ssl.manager.type`.
`vbroker.security.mscapiAliasFix`	`false`	This property was introduced to deal with the Java bug https://bugs.openjdk.java.net/browse/JDK-6483657. This bug, which meant it was not possible to access some certificates in the store when they have the same alias as another certificate, has now been fixed. Rocket Software recommends setting this property to `true` unless it is known that there are definitely no certificates sharing an alias.
`vbroker.security.identityCertificates.nameMustContain`	null	When looking for a client certificate, only consider ones that contain the given string (case-insensitively) in their Friendly Name or Subject Name. Used to assist in picking the right client certificate where you have multiple identity certificates that are otherwise eligible. If this property is not set, and you have multiple certificates that could be sent to the server, the first suitable certificate found in the client's certificate store is used. When using this property you should be aware that this references the full Subject Name rather than the Common Name.

Security Properties for C++

Property	Description
`vbroker.security.logLevel`	Controls the degree of logging. Acceptable values are: `LEVEL_WARN`, `LEVEL_NOTICE`, `LEVEL_INFO`, and `LEVEL_DEBUG`.
`vbroker.security.logFile`	By default, log output is to `std::cerr`. You can use this property to redirect the log output to a named file.
`vbroker.security.secureTransport`	Determines whether 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/TLS. If the server does not support SSL/TLS, this property is set to `false` and the client will not connect.
`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`.
`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`.
`vbroker.security.disable`	If this property is set to `true`, it disables all security services.

Property	Description
`vbroker.security.requireAuthentication`	A server-side only property. Use it to specify whether the client is required to authenticate.

`vbroker.security.authentication.callbackHandler`	Specifies the callback handler for login modules to use for interacting with the user to provide 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.
	For more information, see VisiSecure for C++ APIs.

`vbroker.security.authentication.config`	Specifies the path to the configuration file used for authentication.
`vbroker.security.authentication.retryCount`	Number of times to retry if remote authentication failed.
`vbroker.security.login`	If set to `true`, during initialization the ORB will attempt to log in to all the realms listed by property `vbroker.security.login.realms`.
`vbroker.security.login.realms`	This gives a list of comma-separated realms to log into. This is used when login takes place, either through property `vbroker.security.login` (set to `true`) or API login.

Property	Description
`vbroker.security.vault`	Specifies the path to the vault file. This property takes effect regardless of whether `vbroker.security.login` is set to `true` or `false`.
`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 that 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 (for which the user chooses to provide credentials), 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`.
`vbroker.security.authDomains`	Specifies a comma-separated list of available authorization domains. For example: `vbroker.security.authDomains=domain1,domain2`
`vbroker.security.domain.<domain-name>.rolemap_path`	Specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain `<domain_name>` as specified in: `vbroker.security.authDomains`.
`vbroker.security.domain.<domain_name>.rolemap_enableRefresh`	When set to `true`, enables dynamic loading of the RoleDB file specified by `vbroker.security.domain.<domain_name>.rolemap_path`. The interval of dynamic loading is specified by `vbroker.security.domain.<domain_name>.rolemap_refreshTimeInSeconds`.

Property	Description
`vbroker.security.domain.<domain_name>.rolemap_refreshTimeInSeconds`	Specifies the rolemap refresh time in seconds.
`vbroker.security.domain.<domain_name>.defaultAccessRule`	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`.
`vbroker.security.cert.basicConstraintCritical`	As per the X509 V3 standard, non end-user certificates in a certificate chain must have an extension that is called “basic constraint”. According to the standard, this extension when present must be marked as “critical”, enforcing the recipient to “must understand” and to process accordingly. This enforcement is too strict for an earlier implementation that is aware only of X509 V1.
`vbroker.security.config.root`	The absolute path of the directory to which all relative file system references, for various VisiSecure config files, are based.
`vbroker.security.identityAssertion`	The server side of this ORB is enabled to propagate the IdentityToken sent as part of CSIV2 service context to the next tier (if any). Value can be `true` or `false`.

Property	Description
`vbroker.security.peerAuthenticationMode`	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, and 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 must also 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, they will be authenticated, and 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 must be trusted, otherwise the connection will be refused. `NONE` — Authentication is not required. During SSL/TLS 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.trustpointsRepository`	Specifies a path to the directory containing trusted certificates. The value should be given in the form `Directory:<certs_dir>`. For example: `vbroker.security.trustpointsRepository=Directory:c:\data\identities\Delta`

Property	Description
`vbroker.security.assertions.trust.<n>`	Use 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.assertions.trust.all`	Setting to `true` will trust all assertions made by peers.
`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 to `true`. If `vbroker.security.login.realms` is set, this property is automatically set to `true`. However, you can override it by explicitly setting it in the property file.
`vbroker.security.cipherList`	Set this 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 for use with TLSv1.2 and below.

Property	Description

`vbroker.security.wallet.type`	A wallet is a set of directories containing encrypted private keys and certificate chains for each identity. The possible values are:
	
	`Directory:<path_to_identities>`
	
	Use the `Directory` value to point to the directory containing the directories for all identities.
	
	`PKCS12:<path_to_PKCS#12_KeyStore>/<identity>.p12`
	
	Use the `PKCS12` value to configure the PKCS#12 keystore. See PKCS#12-based authentication using KeyStores for details.

`vbroker.security.wallet.identity`	If `vbroker.security.wallet.type` is set to `Directory`, use this property to point to a sub-directory within the path defined in `vbroker.security.wallet.type` that contains keys and/or certificate information for a specific identity. If `vbroker.security.wallet.type` is set to `PKCS12`, `vbroker.security.wallet.identity` is ignored for a PKCS#12 keystore, but the property must be set.
`vbroker.security.wallet.password`	Specifies the password used to decrypt the private key or the password associated with the login.
`vbroker.security.client.supportNoDelegation`	If set to `true`, the client will add support for NoDelegate in TAG_SSL_SEC_TRANS tag.
`vbroker.security.addOID`	Allows for setting of additional OID.

Property	Description
`vbroker.security.useCAPI`	**Note** VisiBroker for C++ is currently unable to support access to Microsoft certificate stores. As a result of updating to OpenSSL 3.0, VisiBroker for C++'s existing CAPI implementation has become unviable. This is because OpenSSL 1.1 supported CAPI via its Engine API. OpenSSL 3.0 has deprecated the Engine API (in favor of the Provider API). Applications will now find that VisiBroker for C++ will behave as if `vbroker.security.useCAPI` is set to `false`. Rocket Software intends to restore support for MS-CAPI in a future release.
`vbroker.security.useCAPICAs`	**Note** Please refer to `vbroker.security.useCAPI`.
`vbroker.security.useCapiCertificate`	**Note** Please refer to `vbroker.security.useCAPI`.
`vbroker.security.identityCertificates.nameMustContain`	**Note** Please refer to `vbroker.security.useCAPI`.
`vbroker.security.client.socket.allowedDigests`	If set to `short`, this prevents the use of the SHA-2 family of digests (SHA-256, etc).

Property	Description
`vbroker.security.server.socket.TLSSecurityLevel` `vbroker.security.client.socket.TLSSecurityLevel`	OpenSSL now implements a means of tightening the security level in operation. There are six security levels defined; these are described in detail at https://docs.openssl.org/3.0/man3/SSL_CTX_set_security_level/. Starting from level 1, certain cryptographic algorithms and key lengths are disallowed as they are considered too weak for use at that level. Moving up the security levels, the requirements become ever more strict. Level 0 is made available to support legacy behaviors. Note that at level 1, SHA1 and MD5 signatures are disallowed, which in turn has the effect of disallowing TLSv1.0 and TLSv1.1. Any identity (leaf) certificates signed using SHA1 or MD5 will fail unless security level 0 is used. You can override the default by specifying a single integer value from 0 to 5.
`vbroker.security.server.socket.minTLSProtocol` `vbroker.security.client.socket.minTLSProtocol`	Defines the minimum allowable TLS protocol version. These properties and the corresponding `maxTLSProtocol properties` enable you to specify a range of supported TLS protocols, by specifying both a maximum and a minimum supported version. The following values are permitted: - `TLS1` - `TLS1_0` (available to support legacy behaviors) - `TLS_MIN` (a floating minimum, currently equivalent to `TLS1`) - `TLS1_1` - `TLS1_2` - `TLS1_3` - `TLS_MAX` (a floating maximum, currently equivalent to `TLS1_3`) These properties were introduced at VisiBroker 8.5.7; in previous versions it was possible to specify a minimum protocol level, or a single supported protocol (using `vbroker.security.server.socket.enabledProtocols`), but not a range.

Property	Description

`vbroker.security.server.socket.maxTLSProtocol`
 	
`vbroker.security.client.socket.maxTLSProtocol`
	Defines the maximum allowable TLS protocol version.
	
	Used in conjunction with the corresponding `minTLSProtocol` properties, and has the same permitted values.

`vbroker.security.TLS13CipherSuites`	Specifies the cipher suites to be used for TLSv1.3 connections (in preference order). It uses a colon-separated list following the OpenSSL configuration approach. The property defaults to: `TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256` This property should be considered quite separate from `vbroker.security.cipherList`, which continues to be required for configuring cipher suites for TLSv1.2 and below.
`vbroker.security.server.socket.EnforceServerCipherPriority`	This option enables a server to apply its own cipher suite preference order during the TLS handshake. By default, in a TLS handshake sequence the client's cipher suite preference order is honored. However this can leave the server vulnerable to a malicious downgrade attack. To enforce the use of the server's cipher suite preference order, set this property to `true`. This enables a single central cipher suite configuration to control the priority sequence. **Note:** This property defaults to `false` in the current release in order to match previous behavior. It will default to `true` in a future update to increase security.

Property	Description
`vbroker.security.server.socket.MinDHGroupSize`	This is a server-side only option. It applies a minimum limit to the size of DH parameters required to feed into the Diffie-Hellman key exchange during a TLS handshake. This size directly corresponds to the size of the generated ephemeral keys that will be agreed upon and used during the session. VisiSecure supports DH parameter bit sizes of `512`, `1024`, `2048` and `4096`. Any other value specified will be rounded down to the nearest supported value.

`vbroker.security.server.socket.enabledProtocols`
 	
`vbroker.security.client.socket.enabledProtocols`
	Note:
	These properties are deprecated, and will be removed in a future release. Rocket Software recommends using the `maxTLSProtocol` and `minTLSProtocol` properties to specify a range of protocols.
	
	Specifies the SSL protocol version, that is TLSv1.0, TLSv1.1, TLSv1.2 or TLSv1.3, during the SSL/TLS handshaking process when using the VisiSecure, on the server and client sides respectively. Possible values are listed below:
	
	`SSL_Version_3_0_With_2_0_Hello`
	This is a legacy option that enables all available protocol versions.
	
	`SSL_Version_Undetermined`
	This is the default. It allows any of the TLS protocols (TLSv1.0, TLSv1.1, TLSv1.2 or TLSv1.3) and disables SSLv2 or SSLv3 protocols.
	
	`TLS_Version_1_0_With_2_0_Hello`
	In this mode, the library accepts a minimum of TLSv1.0 (currently this results in either TLSv1.0, TLSv1.1,TLSv1.2 or TLSv1.3).
	
	`TLS_Version_1_0_Only`
	Use this option for the highest security when you are certain that the SSL peer also supports TLS 1.0. This option behaves in the same manner as `SSL_Version_3_0_Only`, but applies to the TLS 1.0 protocol.
	
	`TLS_Version_1_1_With_2_0_Hello`
	In this mode, the library accepts a minimum of TLSv1.1 (currently this results in either TLSv1.1, TLSv1.2 or TLSv1.3).
	
	`TLS_Version_1_1_Only`
	In this mode, the library negotiates only a TLSv1.1 connection. However, if the client might also support TLSv1.2 or higher, use `TLS_Version_1_1_With_2_0_Hello` to take advantage of the higher level protocol version.
	
	`TLS_Version_1_2_With_2_0_Hello`
	In this mode, the library accepts a minimum of TLSv1.2 (currently this results in either TLSv1.2 or TLSv1.3)
	
	`TLS_Version_1_2_Only`
	In this mode, the library accepts only a TLSv1.2 connection.

Property	Description

	`TLS_Version_1_3_With_2_0_Hello`
	In this mode, the library accepts a minimum of TLSv1.3. Setting this value future-proofs your configuration should further TLS protocol versions need to be supported.
	
	`TLS_Version_1_3_Only`
	In this mode, the library accepts only a TLSv1.3 connection.

`vbroker.security.CRLRepository`	This property is deprecated. To specify a CRL, use the void addTrustedCertificate (const CORBAsec::X509Cert& trusted,const CORBAsec::ASN1Object* crl = NULL) API instead. Specifies the directory where you want the list of serial numbers of revoked certificates (Certificate Revocation List (CRL)), issued by the Certificate Authority (CA), to reside. All files in the directory will be loaded and interpreted as CRL—no longer valid. 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. For more information, see Certificate Revocation List (CRL) and revoked certificate serial numbers.
`vbroker.security.server.ssl.handshakeTimeout`	Specifies the maximum time (in milliseconds) for the SSL/TLS handshake to complete at the Server side. It can help to prevent the Server from hanging due to unresponsive Clients during the SSL/TLS handshake. The default timeout is 5000ms. To disable the timeout, set it to 0.
`vbroker.security.server.socket.ecdheCurve`	This property is deprecated, and Rocket Software recommends using `TLSCipherGroups` instead. Valid values for this property are identical to those for `vbroker.security.server.socket.TLSCipherGroups`. See note about that property below.

Property	Description

`vbroker.security.server.socket.TLSCipherGroups`
 	
`vbroker.security.client.socket.TLSCipherGroups`
	These options specify the Diffie-Hellman key exchange groups (DH Groups) to be used by server and the client, respectively. Separating server and client configuration allows a single process to have differing DH Groups configured for receiving incoming connections and calling out to remote peers.
	
	Specify as a comma-separated list of curves, each of which must match one of the well-known elliptic curves as defined by IANA (the Internet Assigned Numbers Authority) for use with TLS.
	
	During the TLS handshake, the client and server agree on the group to be used. Groups are specified in preference order, with the most preferred first.
	
	If this option is defined, it overrides the behavior of `vbroker.security.server.socket.ecdheCurve`.
	
	If this option is not defined, it defaults as follows:
	
	1. If `vbroker.security.server.socket.ecdheCurve` is configured (for servers only), its value applies.
	
	2 If not, then the following will apply:
	
	- For TLSv1.3 connections, the `X25519` group is used.
	
	- For TLSv1.2 connections (and below if security level 0 is in operation), the `prime256v1` group is used.

SSL Server Connection Manager properties

The following table lists the SSL Server Connection Manager (SCM) properties.

In this table, possible values for `<se_name>` are:

- `iiop_tp`
- `iiop_ts`

Property	Default	Description
`vbroker.se.<se_name>.scm.ssl.manager.type`	`Socket`	Type of the Server Connection Manager. The only possible value, in VisiBroker for C++, is `Socket`.
`vbroker.se.<se_name>.scm.ssl.manager.connectionMax`	`0`	The maximum number of cached connections on the server. The default value of `0` means that there is no restriction.
`vbroker.se.<se_name>.scm.ssl.manager.connectionMaxIdle`	`0`	Specifies the time, in seconds, which the server uses to determine if an inactive connection should be closed. If a cached connection has been idle longer than this time, then the server closes the connection.
`vbroker.se.<se_name>.scm.ssl.listener.type`	`SSL`	The type of protocol that the listener is using.
`vbroker.se.<se_name>.scm.ssl.listener.port`	`0`	Specifies the port number to be used with the host name property. `0` means that the system will pick a random port number.
`vbroker.se.<se_name>.scm.ssl.listener.proxyPort`	`0`	Specifies the proxy port number to be used with the proxy host name property. `0` means that the system will pick a random port number.

Property	Default	Description
`vbroker.se.<se_name>.scm.ssl.dispatcher.type`	`ThreadPool`	The type of thread dispatcher used in the Server Connection Manager. Possible values are `ThreadPool` and `ThreadSession`.
`vbroker.se.<se_name>.scm.ssl.dispatcher.threadMin`	`0`	Specifies the minimum number of threads that the Server Connection Manager must create.
`vbroker.se.<se_name>.scm.ssl.dispatcher.threadMax`	`0`	Specifies the maximum number of threads that the Server Connection Manager can create.
`vbroker.se.<se_name>.scm.ssl.dispatcher.threadMaxIdle`	`0`	Specifies the time, in seconds, before an idle thread is removed from the thread pool.
`vbroker.se.<se_name>.scm.ssl.connection.tcpNoDelay`	`false`	Specifies whether `tcp_nodelay` should be set on the socket.
`vbroker.se.<se_name>.scm.<scm_name>.listener.selectorMax`	`20`	Specifies the maximum number of NIO Selectors that can be created in the NIO Selector Pool. Increasing this value may improve the throughput if there is a very high number of concurrent invocations. This property is only applicable if NIO SSL Socket is configured at the server side.

Security SPI for C++

This section describes the Service Provider Interface (SPI) classes as defined for 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 plugins must use the `REGISTER_CLASS` macro provided by VisiSecure to register your classes. The name of the class must be specified in full together with its namespace upon registration. The namespace must be specified in a normalized form as 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 modules by extending `vbsec::LoginModule`. To use the login module, you need to set it in the authentication configuration file, just like any other login module.
- **Callback handlers:** You can implement your own callbacks by extending `vbsec::CallbackHandler`. To use the callback, you need to set it in the authentication configuration file, just like any other 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 adapter 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 could specify:

```
vbroker.security.identity.adapters=MyAdapter
vbroker.security.adapter.MyAdapter.property1=value1
vbroker.security.adapter.MyAdapter.property2=value1
```

while an authentication mechanism would provide:

```
vbroker.security.identity.mechanisms=MyMechanism
vbroker.security.adapter.MyMechanism.property1=value1
vbroker.security.adapter.MyMechanism.property2=value2
```

The properties specified will be passed to the user plug-in during initialization as a string map. The map contains truncated key/value pairs 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 plugin 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 scenarios, or can be 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.

Providers

Each provider instance is created by the VisiSecure using a Java reflection API. After the instance has been constructed, the `initialize` method (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 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>`
Authentication Mechanism	`vbroker.security.identity.mechanisms`	`vbsec::AuthenticationMechanisms`	`vbroker.security.identity.mechanism.<name>`
AttributeCodec	`vbroker.security.identity.attributeCodecs`	`vbsec::AttributeCodec`	`vbroker.security.identity.attributeCodec.<name>`

Module Name	Property to set	Interface to implement	Options Prefix
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 of 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 the `ID_ADA1` IdentityAdapter.

Providers and exceptions

During 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

`LoginModule` serves as the parent of all login modules. User-defined 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 for authenticating the given subject and associating relevant Principals and Credentials with the subject. They are also responsible for removing and disposing of such security information during logout.

Include File

`vbauthn.h` 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: 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 might remove any credentials or identities that were established during login and dispose of 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

`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 interactive text mode.

Include file

`vbauthn.h` 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 specific to a mechanism. It is used to perform the decoding and encoding between mechanism-specific and mechanism-independent representations of the entities.

IdentityAdapters included with the VisiSecure

The following IdentityAdapters are provided by 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 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& ) = 0;
```

Returns a Principal representing this identity. This method is used for interfacing with EJBs and servlets.

Arguments

The principal subject.

Returns

A Principal object.

Exceptions

None.

```
virtual ::vbsec::Privileges* getPrivileges( ::vbsec::Subject& ) = 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 direct implementation class 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 could 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

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 Object Format (as defined in `[IETF RFC2743]`).

Returns

A `Target` object.

Exceptions

None.

vbsec::AuthenticationMechanisms

This class represents a fully-fledged mechanism that provides all the functionality required 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` object.
- The corresponding callback handlers used to communicate with user for acquiring the credential.

Returns

A `Subject` containing the acquired credentials (will be null in the case where the operation fails).

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 the 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. One 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
```

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

Another important feature of the Authorization Service is its ability 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 Authorization Service is again 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 logging 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. The Authorization Service makes the decision on whether the subject has permission to access the certain resource based on the privileges object for 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 `::vbsec::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` or `false`

Exceptions

None.

```
std::string toString() const;
```

This method returns “Privileges for ” 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 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 active `AttributeCodec` implementations are registered with the IdentityServices, which is 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 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 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.

```
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` or `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`, encoding 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 `AttributeCodec` (if it 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 privilege 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

`vbsecspishared.h` 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

`vbsecspishared.h` 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` or `false`

Exceptions

None.

```
virtual std::string getActions() const;
```

This method returns the action associated with this `RolePermission`.

Returns

Always returns an empty string, 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 underlying 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. `TrustProvider` 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 behavior 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 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` or `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

`vbsecspishared.h` should be included when you use this class.

Data Members

```
std::map<std::string, std::string>* options;
```

A string map containing name/value pairs 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 increasingly detailed information. There is only one logger available to the whole security service.

Include file

`vbsecspishared.h` 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`.

VisiSecure for C++ APIs

This section describes APIs that are defined in VisiSecure for C++.

All classes are defined with the namespace `vbsec` unless otherwise specified.

Note

HTML documentation of the equivalent VisiSecure for Java API is included in your VisiBroker installation. It can be found at:

`<install_dir>\doc\sec-api-doc`

General API

The general VisiSecure API describes the `Current` and `Context` APIs. It provides API information for Principals, Credentials, and Subjects. In addition, the class vbsec::Wallet API is discussed.

class vbsec::Current

The class `Current` represents the view to the thread-specific security context. This object can be obtained through the following code:

`CORBA::Object_var obj = orb->resolve_initial_references("VBSecurityCurrent");` `Current* c = dynamic_cast <Current*> (obj.in());`

Include File

`vbsec.h` 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 <Context*> (obj.in());`

Include File

`vbsec.h` 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.login.realms`. It traverses the comma-separated 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 callback handler to be used for acquiring information.

`void login (const std::string& realm);`

Login to the system for a specific realm.

Parameter	Description
`realm`	The realm to login to.

`void login (const std::string& realm, vbsec::CallbackHandler& handler);`

Login to the system for a given realm, using a given callback handler for acquiring information.

Parameter	Description
`realm`	The realm to login to.
`handler`	The callback handler to be use for acquiring information.

`void login (const vbsec::Wallet& wallet);`

Login to the system with a `wallet`. Wallets can be created using the `WalletFactory` API.

Parameter	Description
`wallet`	The wallet to be used for login.

`void login (const std::vector<const vbsec::Wallet*>& wallet);`

Login to 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
`wallet`	The Realm for a 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);`

Generates a vault. The vault is written out to the stream that is passed in and encrypted using the password provided (must 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);`

Authenticate 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);`

Sets a seed for the pseudo-random generator used by the SSL/TLS layer.

Parameter	Description
`seed`	The seed for the PRNG.

`ssl::CipherSuiteInfoList* listAvailableCipherSuites();`

Get the list of cipher suites that are available for use with the SSL/TLS layer. Note that this is different from `getEnabledCipherSuites`, in that not all the **available** cipher suites may be currently enabled.

Returns

List of cipher suites that are available (but not necessarily enabled) for use with the SSL/TLS layer.

`void enableCipherSuites (const ssl::CipherSuiteInfoList& suites);`

Sets the cipher suites that should be enabled for all SSL/TLS 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/TLS sessions.

Returns

Cipher suites that are currently enabled for all SSL/TLS sessions.

`void setSSLContext (vbsec::VBSSLContext* ctx);`

Sets the SSL context. This allows the establishment of an SSL/TLS session using the information defined in `VBSSLContext`. A `VBSSLContext` can be created using the `SecureSocketProvider` API.

Parameter	Description
`ctx`	The `VBSSLContext` to be used for SSL/TLS session establishment.

`VBSSLContext& getSSLContext();`

Get the `VBSSLContext` that has been set using `setSSLContext()` or, if not previously set, return the default `VBSSLContext` object.

Returns

The `VBSSLContext` to be used for `SSLSession` establishment.

class vbsec::Principal

Principal represents the identity of a user. This is a virtual class.

Include file

`vbsec.h` 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

`vbsec.h` should be included when you use this class.

class vbsec::Subject

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

`vbsec.h` 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 credentials in the subject. Modifying the content of the set will have no effect on the subject.

`void clearPublicCredentials();`

Clears 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 private credentials in the subject. Modifying the content of the set will have no effect on the subject.

`void clearPrivateCredentials();`

Clears 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 specified by `info`.

Parameter	Description
`info`	The runtime type information used to find matching principals.

Returns

The set of the principals in the subject which have the same runtime information as given. 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 specified by `info`.

Parameter	Description
`info`	The runtime type information used to find matching public credentials.

Returns

The set of the public credentials in the subject which have the same runtime information as given. 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 specified by `info`.

Parameter	Description
`info`	The runtime type information used to find matching private credentials.

Returns

The set of private credentials in the subject which have the same runtime information as given. 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

`vbsec.h` should be included when you use this class.

Methods

`std::string getTarget () const;`

Get the target to which the wallet authenticates.

Returns

The string representation of the target information.

`void populateSubject (Subject& subject);`

Populate the given subject with the necessary credentials, etc. for authentication.

Parameter	Description
`subject`	The subject for the wallet to populate.

class vbsec::WalletFactory

`WalletFactory` is a factory class capable of creating multiple types of wallets.

Include File

`vbsec.h` 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 using a wallet created by this API, the certificate chain will be used in the SSL/TLS 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.
`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.
`passwor`d	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 an identity wallet using a username, password and realm 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 an 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 APIs that interact with VisiSecure's SSL/TLS implementation.

class vbsec::SSLSession

`SSLSession` represents the session of the current SSL/TLS connection. The `SSLSession` can be obtained from `vbsec::Context` using `getPeerSession()`.

Include File

`vbssp.h` should be included when you use this class.

Methods

`time_t getEstablishmentTime() const;`

Get the time when the SSL/TLS connection was established.

Returns

The time when the SSL/TLS connection was established.

`const ssl::CipherSuiteInfo& getNegotiatedCipher() const;`

This method returns the negotiated cipher from the peer for a given SSL/TLS connection.

Returns

The negotiated cipher from the peer for a given SSL/TLS 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.xxx.xxx.xxx.

`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 which will receive the `SSLSession` information.

class vbsec::VBSSLContext

`VBSSLContext` contains information needed to establish an `SSLSession`. This object is created using `SecureSocketProvider::createSSLContext()`.

Include File

`vbssp.h` should be included when you use this class.

Methods

`const CORBAsec::X509CertList& getCertificates() const;`

Gets the certificate chain representing the identity to be used for the SSL/TLS layer.

Returns

The certificate chain representing the identity to be used for the SSL/TLS layer.

`void setCipherSuiteList (const ssl::CipherSuiteInfoList& list);`

This method is used to specify the ciphers available for the SSL/TLS connections.

Parameter	Description
`list`	A list of ciphers that should be available for the SSL/TLS connections.

`const ssl::CipherSuiteInfoList& getCipherSuiteList() const;`

Gets the cipher suites that are currently used by the SSL/TLS layer.

Returns

The cipher suites that are currently used by the SSL/TLS layer.

```
void addTrustedCertificate (const CORBAsec::X509Cert& trusted,
    const CORBAsec::ASN1Object* crl = NULL);
```

Programmatically add a trusted certificate into the SSL context.

Parameter	Description
`trusted`	Certificate that is to be trusted.
`crl`	The CRL issued by the trusted certificate. The 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;`

Gets list of certificates that are currently 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/TLS specification.

Include File

`ssl_c.hh` should be included when you use this class.

class CipherSuiteName

This class provides information about the cipher suites that are available to be used by the Security Service.

Include File

`csstring.h` should be included when you use this class.

Methods

`static const char* toString (int tag);`

Returns a standard representation of a supported SSL/TLS cipher.

Parameter	Description
`tag`	tag associated with the cipher name.

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

`SecureSocketProvider` is the provider for secure socket connections. It provides the function of creating the SSL context, handling SSL/TLS certificates, and managing other secure socket-related information.

Include File

`vbssp.h` 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 creates an SSL context using the given information. The SSL context can then be passed into `vbsec::Context` and used to establish an SSL/TLS connection.

Parameter	Description
`chain`	The certificate chain
`privkey`	The private key object.

Parameter	Description
`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/TLS 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/TLS 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 suites that are available (but not necessarily enabled) for use with the SSL/TLS layer.

`const CertificateFactory& getCertificateFactory() const;`

Gets a certificate Factory.

Returns

A `CertificateFactory` object.

class ssl::Current

`ssl::Current` allows your client application or server object to set its private key and offer its certificate information to its peer. This interface also lets you configure the SSL/TLS connection.

Be aware that PEM-encoded 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 PEM-encoded certificates is:

```
-----BEGIN CERTIFICATE-----
```

- The recognized header line format for PEM-encoded 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 new-line 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 using the following code:

```
CORBA::Object_var obj = orb->resolve_initial_references("SSLCurrent");
ssl::Current_var current = ssl::Current::_narrow(obj);
```

Include File

`ssl_c.hh` should be included when you use this class.

Methods

`CORBA::ULong getNegotiatedCipher(CORBA::Object_ptr peer);`

This method returns the cipher suite that was negotiated during the SSL/TLS handshake with the given peer.

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 the String representation.)

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL/TLS.

`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, although a server can optionally request information from a client.

Parameter	Description
`peer`	The peer from which you wish to obtain their certificate chain.

Returns

List representing the certificate chain offered up by the peer.

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL/TLS.

`char* getPeerAddress(CORBA::Object_ptr peer);`

Returns the IP address of the peer for 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.xxx.xxx.xxx

Exceptions

`CORBA::BAD_OPERATION` if the object is null or the connection is not using SSL/TLS.

`CORBA::Boolean isPeerTrusted(CORBA::Object_ptr peer);`

Tests if the certificate chain of the peer is trusted - that is, if at least one certificate of the chain is in the trustpoint.

Parameter	Description
`peer`	The peer from which you wish to 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/TLS.

`trust::Trustpoints_ptr getTrustpointsObject();`

Use this API to obtain a reference to the trustpoint repository, allowing access to existing trustpoints and for applying additional trustpoints.

Returns

A reference to the trustpoint repository, which should be assigned to a `_var` smart pointer type.

`void setPRNGSeed (const ssl::Current::PRNGseed& seed);`

Sets a seed for the pseudo-random number generator used by the SSL/TLS 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/TLS connections. This is required for servers and is optional for clients. Also look at the `peerAuthenticationMode` property documented in Security Properties for C++.

Parameter	Description
`chain`	The certificate chain, with certificates in DER format
`privkey`	The private key used for the SSL/TLS 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/TLS connections.

Parameter	Description
`list`	A comma-separated list of cipher suites.

`ssl::CipherSuiteInfoList* listAvailableCipherSuites();`

Returns the list of cipher suites that are available in VisiSecure. Note the caller is responsible for freeing the memory associated with the returned data.

Returns

A list of cipher suites.

`ssl::CipherSuiteInfoList* getCipherSuiteList();`

Returns the ciphers that are currently used by the SSL/TLS layer.

Returns

A list of cipher suites.

`void setP12Identity (const ::CORBA::OctetSequence& pks12cert, const char* password);`

Parameter	Description
`pks12cert`	PKCS#12 formatted data.
`password`	Password assocated with the PKCS#12 data file.

Certificate API

class vbsec::CertificateFactory

This is a utility class for handling of certificates and keys.

Include File

`vbssp.h` should be included when you use this class.

Methods

`CORBAsec::X509CertList* importCertificateChain (const CORBAsec::ASN1ObjectList& certs) const;`

Import the certificate chain in the form of `CORBAsec::ASN1ObjectList` into `CORBAsec::X509CertList`, which could be used in `VBSSLContext`.

Parameter	Description
`certs`	`ASN1ObjectList` representation of the certificate chain.

Returns

`CORBAsec::X509CertList` representation of the certificate chain for CORBA transportation.

`CORBAsec::X509CertList* importCertificates (const CORBAsec::ASN1ObjectList& certs) const;`

Import the certificate list in the form of `CORBAsec::ASN1ObjectList` into `CORBAsec::X509CertList`. Certificates need not necessarily 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;
```

Converts a private key from BASE64 or PEM format to DER format.

Parameter	Description
`key`	`ASN1Object` representation of private key object.

Returns

The private key in DER format.

```
CORBAsec::X509CertList* importCertificateChain(
    const CORBAsec::ASN1Object& pkcs12bytes,
    const CSI::UTF8String& password) const;
```

Imports a certificate chain from PKCS#12 binary data.

Parameter	Description
`pkcs12bytes`	`ASN1ObjectList` representation of PKCS#12 data.
`password`	Password for the PKCS#12 data file.

Returns

`CORBAsec::X509CertList` representation of the certificate chain.

```
CORBAsec::ASN1Object* importPrivateKey(
    const CORBAsec::ASN1Object& pkcs12bytes,
    const CSI::UTF8String& password) const;
```

Import private key from PKCS#12 binary data.

Parameter	Description
`pkcs12bytes`	`ASN1ObjectList` representation of PKCS#12 binary data.
`password`	Password for the PKCS#12 data.

Returns

`CORBAsec::ASN1Object` representation of the private key object.

```
const CertificateFactory& printCertificate(
    const CORBAsec::X509Cert& certificate,
    std::ostream& stream) const;
```

Print out the certificate information into an output stream.

Parameter	Description
`certificate`	Certificate whose information is to be printed.

Parameter	Description
`stream`	The 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 X.509 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 also presents a certificate during the SSL/TLS handshake.

Include File

`X509Cert_c.hh` should be included when you use this class.

Methods

`char* getSubjectDN();`

Returns the subject distinguished name (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 distinguished name (DN) contained in the certificate.

Returns

The issuer 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 using the specified algorithm.

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` instances.

Returns

Returns all the extensions available in this certificate as a list of `X509CertExtension` instances. Or, if this certificate has no extensions, the method returns an array of length 0. The extensions are not parsed.

`CORBA::Boolean isValid (CORBA::ULong_out date);`

Verifies that the certificate's 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 X.509 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.
`critical`	Identifies that this extension is 'critical'. Applications that do not recognize a critical extension will not accept the certificate.
`value`	The value of the extension encoded according to the format specified by the oid.

Include File

`X509Cert_c.hh` 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.

Parameter	Description
`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` - `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

`CSIV2Policies.h` should be included when you use this class.

class ServerQoPPolicyImpl

`ServerQoPPolicyImpl` is the concrete implementation of `csiv2::ServerQoPPolicy`. The `ServerQoPPolicyImpl` object impacts the QoP behavior of the server.

Include File

`CSIV2Policies.h` 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();`

Gets the ServerQoPConfigImpl object from the ServerQoPPolicyImpl.

Returns

The `ServerQoPConfigImpl` object from the `ServerQoPPolicyImpl`.

class vbsec::ClientConfigImpl

`ClientConfigImpl` is the concrete implementation of `csiv2::ClientQoPConfig`. To define a `ClientQoPPolicy`, instantiate a `ClientConfigImpl` object to define the various characteristics of the policy.

Include File

`CSIV2Policies.h` 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 `csiv2::ClientQoPPolicy`. The `ClientQoPPolicyImpl` object impacts the QoP behavior of the server.

Include File

`CSIV2Policies.h` 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 used for the policy.

`virtual csiv2::ClientQoPConfig_ptr config();`

Returns the `ClientConfigImpl` object for this `ClientQopPolicyImpl`.

Returns

The `ClientConfigImpl` object for this `ClientQopPolicyImpl`.

Authorization API

The following section describes the classes and methods used for authorization in VisiSecure.

class csiv2::AccessPolicyManager

`AccessPolicyManager` is used to define your Access Policy for authorization of a client's method calls.

Include File

`CSIV2Policies.h` 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& oid,
    const CORBA::OctetSequence& adapter_id);
```

Returns the `ObjectAccessPolicy` for the servant with the given `ObjectId` and object adapter id.

Parameter	Description
`servant`	The CORBA servant object.
`oid`	the ID of the servant object.
`adapter_id`	The ID of the servant object POA.

Returns

`ObjectAccessPolicy` of the servant object.

class csiv2::ObjectAccessPolicy

This class represents the access policy from `AccessPolicyManager`.

Include File

`CSIV2Policies.h` 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.

VisiSecure Error Codes

This appendix provides information about error codes for VisiSecure.

The tables in the subsequent sections list most of the minor codes and their corresponding descriptions that accompany the CORBA system exceptions thrown from within VisiSecure for Java or VisiSecure for C++. The identifier (`ERROR_ID)` helps you to identify or 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_ID`s 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 a brief textual description of the error code.

General Errors

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422100`	`INTERNALNone._ERROR`	Unrecoverable internal runtime error	`NONone._PERMISSION` `BADNone._PARAM` `INTERNAL`
`0x56422101`	`UNTRUSTEDNone._ASSERTION`	Peer has insufficient privilege to assert the caller. See the description of the `vbroker.security.trust` property.	`NONone._PERMISSION`

PKI Errors

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422201`	`ERRORNone._PARSINGNone._CERTIFICATE`	Unknown error when parsing certificate data. Example cause: corrupted DER/Base64 data often happens when copying over FTP that may involve translation for UNIX<->DOS CR-LF (line endings).	`NONone._PERMISSION` `BADNone._PARAM`

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422202`	`ERRORNone._PARSINGNone._PRIVATENone._KEY`	Unknown/unclassified error when parsing private key data. Example cause: corrupted DER/Base64 data often happens when copying over FTP that may involve translation for UNIX<->DOS CR-LF (line endings).	`BADNone._PARAM`

SSL Errors

The following are the translation of SSL Error alerts as defined in the TLS protocol.

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422221`	`UNKNOWNNone._CERTIFICATE`	Translated from SSL alerts: `certificateNone._unknown`	`NONone._PERMISSION` `BADNone._PARAM`
`0x56422222`	`UNSUPPORTEDNone._CERTIFICATE`	Translated from SSL alerts: `unsupportedNone._certificate`	`NONone._PERMISSION`
`0x56422223`	`BADNone._CERTIFICATE`	Translated from SSL alerts: `badNone._certificate`	`NONone._PERMISSION` `BADNone._PARAM`
`0x56422224`	`CERTIFICATENone._REVOKED`	Translated from SSL alerts: `certificateNone._revoked`	`NONone._PERMISSION`

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422225`	`CERTIFICATENone._EXPIRED`	Translated from SSL alerts: `certificateNone._expired`	`NONone._PERMISSION` `BADNone._PARAM`
`0x56422226`	`BADNone._RECORDNone._MAC`	Translated from SSL alerts: `badNone._record None._mac`	`NONone._PERMISSION`
`0x56422227`	`HANDSHAKENone._FAILURE`	Translated from SSL alerts: `handshakeNone._failure`	`NONone._PERMISSION` `BADNone._INVNone._ORDER`

PKCS#12 Errors

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422251`	`P12None._GENERALNone._ERROR`	Unknown error when processing `PKCS#12` data	`BADNone._PARAM` `NONone._RESOURCES`
`0x56422252`	`P12None._INVALIDNone._DATANone._FORMAT`	Corrupted `PKCS#12` data	`BADNone._PARAM`
`0x56422253`	`P12None._INVALIDNone._PKNone._FORMAT`	The private key retrieved from `PKCS#12` data is of unsupported format	`BADNone._PARAM`

Error minor code	Error identifier (ERRORNone._ID)	Error description	Associated CORBA exception
0x56422255	P12None._MISSINGNone._DATA	The PKCS#12 data does not contain the required item	`BADNone._PARAM

General Security Policies (GSP) Errors

Error minor code	Error identifier (ERRORNone._ID)	Error description	Associated CORBA exception
0x56422271	SERVERNone._REQUIRESNone._SECURENone._CONNECTIONS	Secure transport is not available in this client while the server requires it to connect using a secure connection	NONone._PERMISSION
0x56422272	SECURITYNone._CURRENTNone._UNAVAILABLE	SSL session is not available, possible causes: not in a request context, not an SSL connection or invalid object reference (on client side)	BADNone._OPERATION
0x56422273	NONone._POSSIBLENone._CONNECTION	All available connections to the server do not meet the security requirements set up on this client	NONone._PERMISSION

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422274`	`SERVERNone._REQUIRESNone._TRANSPORTNone._IDENTITY`	The server requires transport identity while this client does not have any	`NONone._PERMISSION`

Common Secure Interoperable (CSI) Errors

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422300`	`NONone._IDENTITY`	The server requires at least an identity while this client does not send any	`NONone._PERMISSION`
`0x56422301`	`BADNone._SASNone._DISC`	This client sends SAS (Security Attribute Service, see OMG-CSI) that the server fails to interpret	`NONone._PERMISSION` `BADNone._PARAM`
`0x56422302`	`UPNone._IDENTITYNone._REQD`	The server requires a service context based username/password identity while this client does not send any	`NONone._PERMISSION`
`0x56422303`	`NONone._CONTEXT`	This is the minor code of the `NONone._PERMISSION` exception thrown back to the client carrying `ContextError` with major = 4 and minor = 1 for 'no context' semantic as per OMG-CSI	`NONone._PERMISSION` `TRANSIENT`

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422304`	`CONFLICTINGNone._EVIDENCE`	This is the minor code of the `NONone._PERMISSION` exception thrown back to the clie0nt carrying `ContextError` with major = 3 and minor = 1 for 'conflicting evidence' semantic as per OMG-CSI	`NONone._PERMISSION` `TRANSIENT`
`0x56422305`	`ASSERTIONNone._UNAUTHORIZED`	This is the end-tier that is not supposed to make another call to the next tier on behalf of the caller. See `vbroker.security.supportIdentityAssertion` (Java).	`NONone._PERMISSION`

Authentication Errors

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422400`	`LOGINNone._FAILED`	This is the minor code of the `NONone._PERMISSION` exception thrown back to the client carrying `ContextError` with major = 1 and minor = 1 for 'invalid evidence' semantic as per OMG-CSI	`NONone._PERMISSION` `BADNone._PARAM`

Authorization Errors

Error minor code	Error identifier (`ERRORNone._ID`)	Error description	Associated CORBA exception
`0x56422500`	`FAILEDNone._AUTHORIZATION`	The caller has insufficient privileges required to perform the action	`NONone._PERMISSION`
`0x56422501`	`INVALIDNone._ROLE`	The required run-as role is not configured on this system	`NONone._PERMISSION`
`0x56422502`	`EMPTYNone._ALIASNone._FORNone._ROLE`	The required alias for the specified run-as role is not configured on this system	`NONone._PERMISSION`
`0x56422503`	`ACCESSEDNone._BYNone._UNKNOWNNone._USER`	Access is denied because the caller is unknown. Caller is either null or anonymous	`NONone._PERMISSION`
`0x56422504`	`ACCESSEDNone._BYNone._UNAUTHENTICATEDNone._USER`	Access is denied because the caller is not authenticated	`NONone._PERMISSION`

Login Modules

This appendix provides a description of the LoginModules provided by VisiSecure for Java.

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.

```
<realm-name> {
  com.borland.security.provider.authn.BasicLoginModule <authentication-
requirements-flag>
  DRIVER=<driver-name>
  URL=<database-URL>
  TYPE=basic|tomcat
  LOGINUSERID=<username>
  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 if authenticating to the Tomcat Realm, where they would be required. Otherwise, the remaining properties are sufficient.

Property	Description
`DRIVER`	Fully-qualified class name of the database driver to be used with the password backend. For example, `com.borland.datastore.jdbc.DataStoreDriver`
`URL`	Fully-qualified URL of the database used for the realm.

Property	Description
`nTYPE`	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 default schema. **Note:** If this property is set to `TOMCAT`, all other properties in square brackets (`[..]`) must also be set.
`LOGINUSERID`	Username needed to access the password backend database.
`LOGINPASSWORD`	Password needed to access the password backend database.
`[USERTABLE]`	Table name under which the username/password to be authenticated is stored.
`[USERNAMEFIELDIN-USERTABLE]`	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]`	Table name under which 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 `TYPE` is set to `TOMCAT`, the attributes 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`.

For example:

```
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 passwords should never be stored in clear text, VisiSecure always performs digest on the password and stores the result in a 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 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`

Property	Description
`URL`	Fully-qualified URL of the database used for the password backend.
`[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 which the database stores users.
`USERNAMEFIELD`	The field name in `USERTABLE` containing the usernames.
`ROLETABLE`	Table name under which 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.

For example:

```
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.

Property	Description
`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.
`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 users to type their email when asked for a user name. When set to `DN`, users will type their full `DN` to authenticate themselves.

Property	Description

`QUERY`	The Query option provides a mechanism to dynamically query the LDAP server 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 `HostLoginModule`. For example:

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

UNIX shadow password for Host LoginModule

UNIX platforms only

The `HostLoginModule` shipped with VisiSecure for UNIX platforms utilizes simple APIs that are 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 the POSIX standard, the login module code is less portable.

To write your own custom login module, refer to the 'customlogin' example in the 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 a user database for the basic login module

As a first step, create and configure the database to store users and roles. Rocket Software provides the `userdbadmin` tool, that can be run from the command line to auto-create the required tables, create groups, and associate users with groups.

This example uses JDataStore, though any backend — like Oracle, DB2, Sybase, MS SQL Server, etc — can be used. A sample command is shown below. For JDataStore, the command 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 `&gt;` prompt, create 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 `&gt;` prompt for a list of available commands.

Using the userdbadmin tool

`userdbadmin` is a command-line tool that can be used to create and manage user databases for the `BasicLoginModule`. It uses a proprietary schema and can be pointed at any database. Using this tool, you can administer users who can be authenticated using basic login modules. Though the tool and `BasicLoginModule` work using various JDBC databases, it is still recommended that you use JDataStore which is shipped with VisiBroker.

To facilitate the use of popular databases, the `userdbadmin` tool comes pre-configured to recognize database URLs and to configure itself to use the appropriate drivers.

If you wish to change that behavior, you may override it by specifying the driver information.

If you do not provide driver information and `userdbadmin` does not recognize the database, 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 do not need to type the driver information every time.

Creating a new database

To create a new database, use the 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` in an interactive mode with the created database. The interactive mode helps you to issue multiple commands. To do this, 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 that you are now in `userdbadmin` interactive mode, and you can type commands at its '>' prompt.

Adding new users

To add a new username with password, and to make the new user is a member of one or more groups, type the following at the command line:

Example 1

`> adduser krish krishpwd accountant`

Example 2

`> adduser bill billpwd clerk`

In **Example 1**, you added a user whose name is `Krish` and password `krishpwd` and added him as a member of the group called `accountant`.

In **Example 2**, you added a user whose name is `bill` and password `billpwd` and added him as a member of the group called `clerk`.

Listing existing 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.

`clerk: [ bill ]` `accountant: [ krish john ]`

Create a new group

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 groups `dba` and `admin` would be listed.

```
> listgroups
dba: [ ]
admin: [ ]
clerk: [ bill ]
accountant: [ krish john ]
```

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 memberships would be listed.

```
> listusers
bill: [ clerk ]
john: [ accountant ]
krish: [ accountant dba admin ]
```

Remove a group from the 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 user `jack` would be listed.

```
> listusers
bill: [ clerk ]
jack: [ ]
john: [ ]
krish: [ dba admin ]
```

Remove a group from a user

To remove the group from the user, enter the command `leavegroup` at the command prompt.

`> leavegroups krish admin`

You can check the newly removed user by running the command `listusers`. The user `krish` would be listed without the group from which they were removed.

```
> listusers
bill: [ clerk ]
jack: [ ]
john: [ ]
krish: [ dba ]
```

Remove a user from the group

To remove the user from the group, enter the command `removeuser` at the command prompt.

```
> removeuser bill
```

You can 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 the `userdbadmin` program, enter the command `quit` at the command prompt.

```
> quit
```

Notices

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to www.rocketsoftware.com/support. In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

Country and Toll-free telephone number

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

- United States: 1-855-577-4323
- Australia: 1-800-823-405
- Belgium: 0800-266-65
- Canada: 1-855-577-4323
- China: 400-120-9242
- France: 08-05-08-05-62
- Germany: 0800-180-0882
- Italy: 800-878-295
- Japan: 0800-170-5464
- Netherlands: 0-800-022-2961
- New Zealand: 0800-003210
- South Africa: 0-800-980-818
- United Kingdom: 0800-520-0439