Security Guide

Borland
VisiBroker® 7.0
# Contents

## Chapter 1
**Introduction to Borland VisiBroker** 1
- VisiBroker Overview  
  1
- VisiBroker features  
  2
- VisiBroker Documentation  
  2  
  - Accessing VisiBroker online help topics in the standalone Help Viewer 3  
  - Accessing VisiBroker online help topics from within the VisiBroker Console 3  
  - Documentation conventions 4  
  - Platform conventions 4  
- Contacting Borland support 4  
  - Online resources 5  
  - World Wide Web 5  
  - Borland newsgroups 5

## Chapter 2
**Getting Started with Security** 7  
- VisiSecure overview 8  
  - VisiSecure for Java 8  
  - VisiSecure for C++ 8  
  - Pluggability 8  
  - VisiSecure design flexibility 8  
  - VisiSecure for Java features 8  
  - VisiSecure for C++ Features 9  
- Basic security model 9  
  - Authentication realm (user domain) 10  
  - Resource domain 10  
  - Authorization domain 11  
- Distributed environments and VisiSecure SPI 11  
- Managing authentication and authorization with JAAS 11  
- Authentication and Identification 11  
  - System identification 12  
  - Authentication and pluggability 12  
  - Server and/or client authentication 12  
  - Authenticating clients with usernames and passwords 12  
  - Authentication property settings 13  
  - Public-key encryption 13  
  - Asymmetric encryption 13  
  - Symmetric encryption 14  
  - Certificates and Certificate Authority 14  
  - Digital signatures 14  
  - Generating a private key and certificate request 14  
  - Distinguished names 15  
  - Certificate chains 15  
- Certificate authentication 16  
- Certificate Revocation List (CRL) and revoked certificate serial numbers 16  
- Negotiating Quality of Protection (QoP) parameters 16  
- Secure Transportation 17  
  - JSSE and SSL pluggability 17  
  - Setting the level of encryption 17  
  - Supported cipher suites 17  
- Authorization 18  
  - Access Control List 18  
  - Roles-based access control 18  
  - Pluggable Authorization 19  
- Context Propagation 19  
  - Identity assertions 19  
  - Impersonation 20  
  - Delegation 20  
  - Trusting Assertions 21  
  - Trust assertions and plug-ins 21  
  - Backward trust 21  
  - Forward trust 21  
  - Temporary privileges 21  
- Using IIOP/HTTPS 22  
  - Netscape Communicator/Navigator 22  
  - Microsoft Internet Explorer 22

## Chapter 3
**Authentication** 25  
- JAAS basic concepts 25  
  - Subjects 25  
  - Principals 26  
  - Credentials 26  
  - Public and private credentials 26  
- Authentication mechanisms and LoginModules 27  
  - Authentication realms 27  
  - LoginModules 27  
- LoginContext class and LoginModule interface 28  
- Authentication and stacked LoginModules 29  
- Associating a LoginModule with a realm 30  
  - Syntax of a realm entry 31  
- Borland LoginModules 32  
  - Basic LoginModule 32  
  - JDBC LoginModule 34  
  - LDAP LoginModule 35  
  - Host LoginModule 35  
- Server and Client Identification 36  
  - Setting the config file for client authentication 36  
  - System Identification 36  
  - Formatted Target 37  
  - GSSUP mechanism 37  
  - Certificate mechanism 38  
- Using a Vault 38  
  - Creating a Vault 39  
  - VaultGen example 40  
  - Client identification 41
Chapter 4
Authorization 43
Defining access control with Role DB 43
Anatomy of Role DB 44
Assertion syntax 44
Using logical operators with assertions 45
Wildcard assertions 45
Other assertions 46
Recycling an existing role 46
Authorization domains 46
Run-as Alias 47
Run-as mapping 47
CORBA authorization 48

Chapter 5
Configuring Security Profiles for Domains 51
Security Profiles 51
Enabling Security 53
Enabling SSL 53
Setting the Log Level 54
Configuring Authentication 54
Creating the config.jaas file 54
Configuring Authentication Using the Management Console 55
Configuring Authorization 56
About the rolemap file 56
Configuring Authorization Using the Management Console 56
Specifying VisiSecure properties 62
Associating a Profile with a Domain 63
Using a Vault for a Domain 63

Chapter 6
Making Secure Connections (Java) 65
JAAS and JSSE 65
JSSE Basic Concepts 65
Steps to secure clients and servers 66
Step One: Providing an identity 66
Username/password authentication, using JAAS modules, for known realms 66
Username/password authentication, using APIs 67
Certificate-based authentication, using KeyStores through property settings 67
Certificate-based authentication, using KeyStores through APIs 67
Certificate-based authentication, using APIs 67
pkcs12-based authentication, using KeyStores 67
pkcs12-based authentication, using APIs 67
Step Two: Setting properties and Quality of Protection (QoP) 68
Step Three: Setting up Trust 68
Step Four: Setting up the Pseudo-Random Number Generator 68
Step Five: If necessary, set up identity assertion 68
Examining SSL related information 69
Creating Custom Plugins 69
LoginModules 69
CallbackHandlers 69
Authorization Service Provider 69
Trust Providers 70

Chapter 7
Making Secure Connections (C++) 71
Steps to secure clients and servers 71
Step One: Providing an identity 71
Username/password authentication, using LoginModules, for known realms 72
Username/password authentication, using APIs 72
Certificate-based authentication, using KeyStores through property settings 72
Certificate-based authentication, using KeyStores through APIs 72
Certificate-based authentication, using APIs 73
pkcs12-based authentication, using KeyStores 73
pkcs12-based authentication, using APIs 73
Step Two: Setting properties and Quality of Protection (QoP) 73
Step Three: Setting up Trust 73
Step Four: If necessary, set up identity assertion 74
Examining SSL related information 74
Creating Custom Plugins 74
LoginModules 75
CallbackHandlers 75
Authorization Service Provider 75
Trust Providers 76

Chapter 8
Security for the Web components 77
Security for the Apache web server 77
Modifying the Apache configuration file for mod_ssl 77
Creating key and certificate files 79
Verifying that mod_ssl is active 80
Enabling certificate passthrough to the Borland web container 81
Configuring Apache to "passthrough" the SSL certificate and related information 81
Configuring the mod_iop IIOP connector of the httpd.conf file to forward SSL authentication 82
Chapter 9
Security Properties for Java 87

Chapter 10
Security Properties for C++ 91

Chapter 11
VisiSecure for C++ APIs 95

General API
class vbsec::Current 95
Include File 95
Methods 95
class vbsec::Context 96
Include File 96
Methods 96
class vbsec::Principal 99
Include file 99
Methods 99
class vbsec::Credential 99
Include File 99
class vbsec::Subject 100
Include File 100
Methods 100
class vbsec::Wallet 101
Include File 101
Methods 101
class vbsec::WalletFactory 101
Include File 101
Methods 101
SSL API 103
class vbsec::SSLSocket 103
Include File 103
Methods 103
class vbsec::VBSSLContext 104
Include File 104
Methods 104
class ssl::CipherSuiteInfo 104
Include File 104
class CipherSuiteName 105
Include File 105
Methods 105
class vbsec::SecureSocketProvider 105
Include File 105
Methods 105
class ssl::Current 106
Include File 106
Methods 106
Certificate API 109
class vbsec::CertificateFactory 109
Include File 109
Methods 109

class CORBAsec::X509Cert 111
Include File 111
Methods 111
class CORBAsec::X509CertExtension 112
Include File 112
QoP API 113
class vbsec::ServerConfigImpl 113
Include File 113
Methods 113
class ServerQoPConfigImpl 113
Include File 113
Methods 113
class vbsec::ClientConfigImpl 114
Include File 114
Methods 114
class vbsec::ClientQoPConfigImpl 114
Include File 114
Methods 114
Authorization API 115
class csiv2::AccessPolicyManager 115
Include File 115
Methods 115
class csiv2::ObjectAccessPolicy 116
Include File 116
Methods 116

Chapter 12
Security SPI for C++ 117

Plugin Mechanism and SPIs 117
Providers 119
Providers and exceptions 119
vbsec::LoginModule 120
Include File 120
Methods 120
vbsec::CallbackHandler 121
Include file 121
Methods 121
vbsec::IdentityAdapter 121
IdentityAdapters included with the VisiSecure 121
Methods 121
vbsec::MechanismAdapter 122
Methods 123
vbsec::AuthenticationMechanisms 123
Credential-related methods 123
Context-related methods 124
vbsec::Target 125
Methods 125
vbsec::AuthorizationServiceProvider 126
Methods 126
vbsec::Resource 127
Methods 127
vbsec::Privileges 127
Constructors 127
Methods 127
vbsec::AttributeCodec 128
Methods 128
Introduction to Borland VisiBroker

For the CORBA developer, Borland provides VisiBroker for Java, VisiBroker for C++, and VisiBroker for .NET to leverage the industry-leading VisiBroker Object Request Broker (ORB). These three facets of VisiBroker are implementations of the CORBA 2.6 specification.

VisiBroker Overview

VisiBroker is for distributed deployments that require CORBA to communicate between both Java and non-Java objects. It is available on a wide range of platforms (hardware, operating systems, compilers and JDKs). VisiBroker solves all the problems normally associated with distributed systems in a heterogeneous environment.

VisiBroker includes:

- VisiBroker for Java, VisiBroker for C++, and VisiBroker for .NET, three implementations of the industry-leading Object Request Broker.
- VisiNaming Service, a complete implementation of the Interoperable Naming Specification in version 1.3.
- GateKeeper, a proxy server for managing connections to CORBA Servers behind firewalls.
- VisiBroker Console, a GUI tool for easily managing a CORBA environment.
- Common Object Services such as VisiNotify (implementation of Notification Service Specification), VisiTransact (implementation of Transaction Service Specification), VisiTelcoLog (implementation of Telecom Logging Service Specification), VisiTime (implementation of Time Service Specification), and VisiSecure.
VisiBroker offers the following features:

- “Out-of-the-box” security and web connectivity.
- Seamless integration to the J2EE Platform, allowing CORBA clients direct access to EJBs.
- A robust Naming Service (VisiNaming), with caching, persistent storage, and replication for high availability.
- Automatic client failover to backup servers if primary server is unreachable.
- Load distribution across a cluster of CORBA servers.
- Full compliance with the OMG’s CORBA 2.6 Specification.
- Integration with the Borland JBuilder integrated development environment.
- Enhanced integration with other Borland products including Borland AppServer.

VisiBroker documentation set includes the following:

- Borland VisiBroker Installation Guide—describes how to install VisiBroker on your network. It is written for system administrators who are familiar with Windows or UNIX operating systems.
- Borland VisiBroker for Java Developer’s Guide—describes how to develop VisiBroker applications in Java. It familiarizes you with configuration and management of the Visibroker ORB and how to use the programming tools. Also described is the IDL compiler, the Smart Agent, the Location, Naming and Event Services, the Object Activation Daemon (OAD), the Quality of Service (QoS), the Interface Repository, and the Interface Repository, and Web Service Support.
- Borland VisiBroker for C++ Developer’s Guide—describes how to develop VisiBroker applications in C++. It familiarizes you with configuration and management of the Visibroker ORB and how to use the programming tools. Also described is the IDL compiler, the Smart Agent, the Location, Naming and Event Services, the OAD, the QoS, Pluggable Transport Interface, RT CORBA Extensions, and Web Service Support.
- Borland VisiBroker for C++ API Reference—provides a description of the classes and interfaces supplied with VisiBroker for C++.
- Borland VisiBroker VisiTime Guide—describes Borland’s implementation of the OMG Time Service specification.
- Borland VisiBroker VisiNotify Guide—describes Borland’s implementation of the OMG Notification Service specification and how to use the major features of the notification messaging framework, in particular, the Quality of Service (QoS) properties, Filtering, and Publish/Subscribe Adapter (PSA).
- Borland VisiBroker VisiTelcoLog Guide—describes Borland's implementation of the OMG Telecom Log Service specification.
- Borland VisiBroker GateKeeper Guide—describes how to use the VisiBroker GateKeeper to enable VisiBroker clients to communicate with servers across networks, while still conforming to the security restrictions imposed by web browsers and firewalls.

The documentation is typically accessed through the Help Viewer installed with VisiBroker. You can choose to view help from the standalone Help Viewer or from within a VisiBroker Console. Both methods launch the Help Viewer in a separate window and give you access to the main Help Viewer toolbar for navigation and printing, as well as access to a navigation pane. The Help Viewer navigation pane includes a table of contents for all VisiBroker books and reference documentation, a thorough index, and a comprehensive search page.

Important Updates to the product documentation, as well as PDF versions, are available on the web at [http://www.borland.com/techpubs](http://www.borland.com/techpubs).

### 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**
  - Choose Start|Programs|Borland Deployment Platform|Help Topics
  - or, open the Command Prompt and go to the product installation \bin directory, then type 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 the VisiBroker Console

To access the online help from within the VisiBroker Console, choose 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.
Documentation conventions

The documentation for VisiBroker uses the typefaces and symbols described below to indicate special text:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>italics</td>
<td>Used for new terms and book titles.</td>
</tr>
<tr>
<td>computer</td>
<td>Information that the user or application provides, sample command lines and code.</td>
</tr>
<tr>
<td>bold computer</td>
<td>In text, bold indicates information the user types in. In code samples, bold highlights important statements.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional items.</td>
</tr>
<tr>
<td>...</td>
<td>Previous argument that can be repeated.</td>
</tr>
<tr>
<td></td>
<td>Two mutually exclusive choices.</td>
</tr>
</tbody>
</table>

Platform conventions

The VisiBroker documentation uses the following symbols to indicate platform-specific information:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>All supported Windows platforms.</td>
</tr>
<tr>
<td>Win2003</td>
<td>Windows 2003 only</td>
</tr>
<tr>
<td>WinXP</td>
<td>Windows XP only</td>
</tr>
<tr>
<td>Win2000</td>
<td>Windows 2000 only</td>
</tr>
<tr>
<td>UNIX</td>
<td>UNIX platforms</td>
</tr>
<tr>
<td>Solaris</td>
<td>Solaris only</td>
</tr>
<tr>
<td>Linux</td>
<td>Linux only</td>
</tr>
</tbody>
</table>

Contacting Borland support

Borland offers a variety of support options. These include free services on the Internet where you can search our extensive information base and connect with other users of Borland products. In addition, you can choose from several categories of telephone support, ranging from support on installation of Borland products to fee-based, consultant-level support and detailed assistance.

For more information about Borland’s support services or contacting Borland Technical Support, please see our web site at: http://support.borland.com and select your geographic region.

When contacting Borland’s support, be prepared to provide the following information:

- Name
- Company and site ID
- Telephone number
- Your Access ID number (U.S.A. only)
- Operating system and version
- Borland product name and version
- Any patches or service packs applied
Contacting Borland support

- Client language and version (if applicable)
- Database and version (if applicable)
- Detailed description and history of the problem
- Any log files which indicate the problem
- Details of any error messages or exceptions raised

**Online resources**

You can get information from any of these online sources:

<table>
<thead>
<tr>
<th>World Wide Web</th>
<th><a href="http://www.borland.com">http://www.borland.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Online Support</td>
<td><a href="http://support.borland.com">http://support.borland.com</a> (access ID required)</td>
</tr>
<tr>
<td>Listserv</td>
<td>To subscribe to electronic newsletters, use the online form at: <a href="http://www.borland.com/products/newsletters">http://www.borland.com/products/newsletters</a></td>
</tr>
</tbody>
</table>

**World Wide Web**

Check http://www.borland.com/bes regularly. The VisiBroker Product Team posts white papers, competitive analyses, answers to FAQs, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- http://www.borland.com/techpubs (documentation updates and PDFs)
- http://info.borland.com/devsupport/bdp/faq/ (VisiBroker FAQs)
- http://community.borland.com (contains our web-based news magazine for developers)

**Borland newsgroups**

You can participate in many threaded discussion groups devoted to the Borland VisiBroker. Visit http://www.borland.com/newsgroups for information about joining user-supported newsgroups for VisiBroker and other Borland products.

**Note** These newsgroups are maintained by users and are not official Borland sites.
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 aVisiSecure 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 for Java

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

VisiSecure for C++


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 design flexibility

Borland has designed VisiSecure 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, alone it cannot fully protect your servers. You must be responsible for physical security, and configuring you base web server (host) and operating system services in the most secure manner possible.

VisiSecure for Java features

VisiSecure has the following features:

- **Enterprise Java Beans (EJB) Container Integration**: VisiSecure seamlessly integrates EJB security mechanisms with the underlying CORBA Security Service and CSIV2. CORBA offers enhanced features to the security architecture of your bean. By utilizing VisiSecure, you have additional options over the relatively simple EJB security model.
**Basic security model**

- **Web Container Integration**: VisiSecure integrates with the web container by providing mechanisms to the web container that allow its own authentication and authorization engines to propagate security information to other EJB containers, as necessary. For example, a servlet trying to invoke an EJB container's bean will act on behalf of the original browser client that triggered the initial request. Security information supplied from the client will be propagated seamlessly to the EJB container. In addition, the web container authentication and authorization engine can be configured to use authentication LoginModules and authorization rolemaps supplied by Borland.

- **Security Services Administrator**: The administration and configuration of VisiSecure is performed using simple-to-use properties and supports tools like the Java keytool.

- **GateKeeper**: You can use GateKeeper to enable authenticated connections across a high-level firewall. This allows clients to connect to the server, even if the server and the application client are on opposite sides of a firewall. Use of the GateKeeper is fully documented in the *VisiBroker GateKeeper Guide*.

- **Secure Transport Layer**: VisiSecure utilizes SSL, the primary secure transport level communication protocol on the Internet, as a secure transport layer. SSL provides message confidentiality, message integrity, and certificate-based authentication support through a trust model.

**VisiSecure for C++ Features**

VisiSecure for C++ has the following features:

- **Authentication and Authorization**: The Authentication and Authorization model are similar to VisiSecure for Java. This extends the capability of VisiSecure for C++ applications.

- **Security Services Administrator**: The administration and configuration of VisiSecure is performed using simple-to-use properties.

- **Secure Transport Layer**: VisiSecure utilizes SSL, the primary secure transport-level communication protocol on the Internet, as a secure transport layer. SSL provides message confidentiality, message integrity, and certificate-based authentication support through a trust model.

**Basic security model**

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

- **Authentication realm (User domain)**: 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.

**Figure 2.1 Interaction Among Different Domains in VisiSecure**

![Interaction Among Different Domains in VisiSecure](image)

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.

An authentication realm is defined both by the authentication technology it uses, as well as a set of configuration options that point to the source of the data. 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, see Chapter 3, “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 servlet 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 Chapter 4, “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 Chapter 4, “Authorization.”

**Distributed environments and VisiSecure SPI**

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

**Managing authentication and authorization 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.

**Authentication and Identification**

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

1. acquiring credentials from the to-be-authenticated entity,
2. then verifying the credentials.

VisiSecure employs the JAAS framework to facilitate the interaction between the entities and the system.
**System identification**

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

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

For more information, see “System Identification” on page 36.

**Authentication and pluggability**

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

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

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

**Server and/or client authentication**

With the VisiBroker implementation of JAAS, you can set different mechanisms of authentication. You can have server authentication, where servers are authenticated by clients using public-key certificates. You can also have client authentication. Clients can be authenticated using passwords or public-key certificates. That is, the server can be configured to authenticate clients with a password or clients with public-key certificates.

**Authenticating clients with usernames and passwords**

If server-side authentication is not required, authentication can be accomplished using a standard username/password combination. 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 more information, see Chapter 3, “Authentication.”
Authentication property settings

The authentication policy—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 Chapter 10, “Security Properties for C++” and Chapter 9, “Security Properties for Java.”

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. Note that encryption can be asymmetric or symmetric.

Asymmetric encryption

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

Symmetric encryption uses only one key for both encryption and decryption. Although faster than asymmetric encryption, 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.

Digital signatures

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

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

Generating a private key and certificate request

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

After you generate the files, you should submit the certificate request to a CA. The procedure for submitting your certificate request to a CA is determined by the certificate authority which you are using. If you are using a CA that is internal to your organization, contact your system administrator for instructions. If you are using a commercial CA, you should contact them for instructions on submitting your certificate request. The certificate request you send to the CA will contain your public key and your distinguished name.
Distinguished names

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

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

Certificate chains

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

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

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

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

Closely associated with authentication is the concept of trust. For practical purposes, trust operates just like authentication. Trust can be applied at the transport level if a certificate identity is presented, or at even higher levels (at the CSIv2 layer) where the identity takes the form of a username/password.

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

C++ For VisiBroker for C++ users, the a set of APIs that allow trustpoints (trusted certificates) to be configured is provided as well. For more information, see Chapter 11, “VisiSecure for C++ APIs.”

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 (CAs), see the “Certificates and Certificate Authority” on page 14.

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

The CRL files must be in DER format and are stored in a directory. To input a CRL into VisiSecure for C++, you need to set the vbroker.security.CRLRepository property to the directory where the CRL files reside. For more information on VisiSecure for C++ properties, see Chapter 10, “Security Properties for C++.”

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

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

Negotiating 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” on page 113.
Secure Transportation

VisiSecure functions in two transport environments:

- using IIOP over plain sockets
- using secure sockets (SSL)

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 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).

JSSE and SSL pluggability

Java VisiSecure uses Java Secure Sockets Extension (JSSE) to perform SSL communication. VisiSecure SPI Secure Socket Provider class provides access to the underline SSL implementation. Any appropriate implementation following Java Secure Socket Extension (JSSE) framework can be easily plugged in independent of other provider mechanisms. The only necessary step is mapping the interfaces (or, in another word, callback methods) defined to the corresponding JSSE implementation. For more information on the SPI Secure Socket Provider class, see VisiSecure SPI for Java and Chapter 12, “Security SPI for C++.”

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

Setting the level of encryption

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

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

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

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

Supported cipher suites

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

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

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

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

Authorization

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

Access Control List

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

Roles-based access control

VisiSecure uses an access control scheme based on roles. The deployment descriptor maintains a list of roles that are authorized to access each enterprise bean method. VisiSecure uses a role database (a file whose default name is `roles.db`) to do the association between user identities and EJB roles. If a user is associated with at least one role, the user may access the method. For more information, see Chapter 4, “Authorization.”
**Pluggable Authorization**

VisiSecure provides the ability to plug-in an authorization service that can map users to roles. The implementer of the Authorization Service provides the collection of permission objects granted access to certain resources. A new class, `RolePermission`, is defined to represent “role” as permission. The 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.

The Authorization Service is tightly connected with the concept of Authorization domain—each domain has exactly one Authorization Services Provider implementation. The Authorization domain is the bridge between VisiSecure system and the authorization service implementation. During the initialization of the ORB itself, the authorization domains defined by the property `vbroker.security.authDomains` are constructed, while the Authorization Services Provider implementation is instantiated during the construction of the Authorization domain itself.

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

For more information, see Chapter 4, “Authorization.”

**Context Propagation**

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

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

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

**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.
Trust assertions and plug-ins

When a remote peer (server or process) makes identity assertions while acting on behalf of the callers, the end-tier server needs to trust the peer to make such assertions. The Security Provider Interface (SPI) allows you to plug in a Trust Services Provider to determine whether the assertion is allowed (trusted) for a given caller and a given set of privileges for the asserter. Specifically, you use the TrustProvider class to implement trust rules that determine whether the server will accept identity assertions from a given asserting subject. For more information, see sec-api-doc in the Help system, and the Chapter 12, “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 effective requires to assume an identity that has access to all resources that require the user to be in that role.
Using IIOP/HTTPS

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

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

Netscape Communicator/Navigator

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

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

Microsoft Internet Explorer

To use IIOP/HTTPS with Microsoft Internet Explorer, you must make sure that the HTTPS connection requires no user interaction. For example, if the browser visits a HTTPS site with an untrusted root certificate, the browser will ask for permission before establishing an HTTPS connection. The Microsoft JVM, due to a known bug, fails on this connection.
Here are several examples that illustrate this condition and ways in which you can work:

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

Internet Explorer optionally requires the Common Name field within the server certificate to be the same as the host name of the server. From the View|Internet Options menu, select the Advanced tab and scroll to the Security section. Make sure the box next to Warn about invalid site certificates is not checked to use a server certificate that does not contain the host name of the server.
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.

**JAAS basic concepts**

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

**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, such as in the code sample below:

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

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

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

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

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

Credentials

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

Public and private credentials

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

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

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

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

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

C++

```cpp
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. No permission required to access both public and private credentials. For more information on permissions in Java, consult the JAAS Specification from Sun Microsystems.

### 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 Security Provider Interface classes for Java and C++ that enable you to “plug-in” security service provider implementations of authentication and identification.

### Authentication realms

An **authentication realm** represents a single user authentication mechanism, customized to point to a datasource which contains user information. This allows the authentication mechanism to be independent of the actual user database and therefore be used with multiple user databases that support the same authentication mechanism. For example, if a vendor writes an authentication module to work with LDAP, that mechanism can then be used to interact with different LDAP directories in different environments, without having to rewrite or otherwise modify the authentication mechanism.

For more information on the authentication realm (user domain), see “Basic security model” on page 9.

### 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” on page 30.
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

The Borland VisiBroker Server 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:

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

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

![Figure 3.1 Realm entry in a JAAS config](image)

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

**Note** There must be at least one LoginModule with the authentication requirements flag = required.
Associating a LoginModule with a realm

Syntax of a realm entry

Each realm entry has a particular syntax that must be followed. The following code sample shows the generic syntax for a realm entry:

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

Note

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

The following four elements are found in the realm entry:

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

Borland provides the following common LoginModules for server and client authentication. These LoginModules are used for both client authentication and authentication of the Borland VisiBroker 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.

- **BasicLoginModule**—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++.

### 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.

```plaintext
realm-name {
    com.borland.security.provider.authn.BasicLoginModule authentication-requirements-flag
    DRIVER=driver-name
    URL=database-URL
    TYPE=basic|tomcat
    LOGINUSERID=user-name
    LOGINPASSWORD=password
    [USERTABLE=user-table-name]
    [GROUPTABLE=group-table-name]
    [GROUPNAMEFIELD=group-name-field-of-GROUPTABLE]
    [PASSWORDFIELD=field-name]
    [USERNAMEFIELDINUSERTABLE=field-name]
    [USERNAMEFIELDINGROUPTABLE=field-name]
    [DIGEST=digest-name]
};
```
Borland LoginModules

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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRIVER</td>
<td>Fully-qualified class name of the database driver to be used with the password backend. For example, com.borland.datastore.jdbc.DataStoreDriver</td>
</tr>
<tr>
<td>URL</td>
<td>Fully-qualified URL of the database used for the realm.</td>
</tr>
<tr>
<td>TYPE</td>
<td>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 &quot;TOMCAT&quot; to use the Tomcat schema. Set this to &quot;basic&quot; to use the Borland schema. <strong>Note:</strong> If this property is set to &quot;TOMCAT,&quot; all other properties in square braces (&quot;[..]&quot;) must also be set.</td>
</tr>
<tr>
<td>LOGINUSERID</td>
<td>Username needed to access the password backend database.</td>
</tr>
<tr>
<td>LOGINPASSWORD</td>
<td>Password needed to access the password backend database.</td>
</tr>
<tr>
<td>[USERTABLE]</td>
<td>Table name under which the username/password to be authenticated is stored.</td>
</tr>
<tr>
<td>[USERNAMEFIELDINUSER-TABLE]</td>
<td>The field name in USERTABLE where the userID can be read.</td>
</tr>
<tr>
<td>[USERNAMEFIELDINGROUPTABLE]</td>
<td>The field name in GROUPTABLE where the userID can be read, different from that in the USERTABLE.</td>
</tr>
<tr>
<td>[PASSWORDFIELD]</td>
<td>The field name in USERTABLE containing the password for the username to be authenticated.</td>
</tr>
<tr>
<td>[GROUPTABLE]</td>
<td>Table name under where the group information for the user is stored. When TYPE is set to &quot;TOMCAT,&quot; the attribute represented by entries in this table are treated as roles rather than groups.</td>
</tr>
<tr>
<td>[GROUPNAMEFIELD]</td>
<td>Name of the field in GROUPTABLE containing the group name to be associated with the user. When TYPE is set to &quot;TOMCAT,&quot; the attribute represented by entries in this table are treated as roles rather than groups.</td>
</tr>
<tr>
<td>[DIGEST]</td>
<td>The algorithm to use for digesting the password. This defaults to SHA under basic circumstances, but defaults to MD5 when TYPE is set to &quot;TOMCAT&quot;.</td>
</tr>
</tbody>
</table>

Premium {
    com.borland.security.provider.authn.BasicLoginModule required
    DRIVER="com.borland.datastore.jdbc.DataStoreDriver"
    URL="jdbc:borland:dslocal:/Security/java/prod/userauthinfo1.jds"
    Realm="Basic"
    LOGINUSERID="CreateTx"
    LOGINPASSWORD="";
}

Since password should never be stored in clear text, VisiSecure always performs digest on the password and stores the result into 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.

```plaintext
realm-name {
    com.borland.security.provider.authn.JDBCLoginModule authentication-requirements-flag
    DRIVER=driver-name
    URL=database-URL
    [DBTYPE=type]
    USERTABLE=user-table-name
    USERNAMESFIELD=user-name-field-of-USERTABLE
    ROLETABLE=role-table-name
    ROLENAMEFIELD=role-name
    USERNAMEFIELDINROLE-TABLE=role-name
    USERNAME=
    PASSWORD=
};
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRIVER</td>
<td>Fully-qualified class name of the database driver to be used with the realm. For example, com.borland.datastore.jdbc.DataStoreDriver</td>
</tr>
<tr>
<td>URL</td>
<td>Fully-qualified URL of the database used for the password backend.</td>
</tr>
<tr>
<td>[DBTYPE=] ORACLE</td>
<td>SYBASE</td>
</tr>
<tr>
<td>USERTABLE</td>
<td>Table name under where the database stores users.</td>
</tr>
<tr>
<td>USERNAMESFIELD</td>
<td>The field name in USERTABLE containing the usernames.</td>
</tr>
<tr>
<td>ROLETABLE</td>
<td>Table name where the database stores the roles of users.</td>
</tr>
<tr>
<td>ROLENAMEFIELD</td>
<td>Field name in ROLETABLE where role information is stored.</td>
</tr>
<tr>
<td>USERNAMESFIELDINROLE-TABLE</td>
<td>The username field name in the ROLETABLE.</td>
</tr>
<tr>
<td>USERNAME</td>
<td>The username needed to access the password backend database.</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>The password needed to access the password backend database.</td>
</tr>
</tbody>
</table>

```plaintext
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
    USERNAMESFIELD=userNames
    ROLETABLE=roleTable
    ROLENAMEFIELD=roleNames
    USERNAMESFIELDINROLE-TABLE=roleName
    USERNAME="\n"
    PASSWORD="\n";
};
```
## LDAP LoginModule

Similar to the JDBC LoginModule, but using LDAP as its authentication backend.

```plaintext
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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALCONTEXTFACTORY</td>
<td>The InitialContextFactory class that is used by JNDI to bind to LDAP.</td>
</tr>
<tr>
<td>PROVIDERURL</td>
<td>The URL to the LDAP server of the form ldap://&lt;servername&gt;:&lt;port&gt;.</td>
</tr>
<tr>
<td>SEARCHBASE</td>
<td>The search base for the Directory to lookup.</td>
</tr>
<tr>
<td>USERATTRIBUTES</td>
<td>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.</td>
</tr>
<tr>
<td>USERNAMEATTRIBUTE</td>
<td>This attribute represents what the user types in as the username. If set to <code>uid</code> it would allow users to type their <code>uid</code> when asked for a username. If set to <code>mail</code>, it would allow the user to type their email when asked for a user name. When set to <code>DN</code> the user will types their full <code>DN</code> to authenticate themselves.</td>
</tr>
<tr>
<td>QUERY</td>
<td>The Query options provides a mechanism to dynamically query the LDAP for other information and represent the results as attributes. For example, a user can be a member of a set of groups. It is useful to extract this information as the GROUP attribute so that it can be used in rules in the authorization domain. To achieve this, you can specify a query. Queries are of the format:</td>
</tr>
</tbody>
</table>

```plaintext
query.<suffix>="<attname>=<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:

```plaintext
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.

```plaintext
realm-name { 
    com.borland.security.provider.authn.HostLoginModule authentication-requirements-flag;
};
```
No additional properties are necessary for the Host LoginModule.

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

---

**Server and Client Identification**

In addition to the many clients and users that need to be authenticated to the various VisiBroker services, the Borland VisiBroker Server 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. In general, any system that needs to engage in secure communication as a client, must be configured to have an identity that represents the user/client on whose behalf it is acting. When using SSL with mutual authentication, a server also needs a certificate to identify itself to the client.

---

**Setting the config file for client authentication**

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

To set the location of the configuration file:

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

---

**System Identification**

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

For example:

Set the property `vbroker.security.login=true`

Set the property `vbroker.security.login.realms=payroll,hr`

Set the following realm information in a file reference by `vbroker.security.authentication.config=<config-file>`

Set the property `vbroker.security.callbackhandler=<callback-handler>`

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

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

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

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

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

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

This format is called **Formatted Target**.

### Formatted Target

A “realm” represents a configuration entry that represents an authenticating target. In the absence of a configuration file (such as in a client process, or for certificates, which have no representation in a JAAS config file), there needs to be a way to represent a target realm. This is done using a “formatted target”. A formatted target is of the form:

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

For example:

- `Realm1`, `Realm3`, `GSSUP#Realm4`, and `Certificate#ALL`.

An authentication mechanism represents a “format” for communicating authentication information between various components of the security subsystem. For example, it represents how LoginModules communicate with the mechanism and how the mechanism on one process communicates with an equivalent mechanism on another process. The mechanism specific target name represents how the mechanism represents this target.

**GSSUP mechanism**

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

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

```
GSSUP#ServerRealm
```

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

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

You can put certificates into VisiSecure using certificate login or wallet APIs. When using wallet APIs, you need to specify the usage through the constant definitions in the \texttt{vbsec.h} file, class \texttt{vbsec::WalletFactory}. For more information, see “class \texttt{vbsec::WalletFactory}” on page 101.

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

\texttt{Certificate<target>}

\textbf{Note}  
If you do not specify the usage, the default is \texttt{ALL}.

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

\begin{table}[h]
\centering
\begin{tabular}{|c|p{0.7\textwidth}|}
\hline
\textbf{Target} & \textbf{Description} \\
\hline
Certificate\#CLIENT & Identifies this process in a client role. When a user establishes an identity for this target, the certificate identity established will be used when this process acts as a client. In other words, this certificate will identify this process when it establishes outgoing SSL connections. \\
\hline
Certificate\#SERVER & Identifies this process in a server role. When a user establishes an identity for this target, this process will use the certificate identity established to identify itself when it is accepting SSL connections. \\
\hline
Certificate\#ALL & Identifies this process in all roles. This identity is used in both of the above roles. \\
\hline
\end{tabular}
\caption{Targets Defined for the Certificate Login Mechanism}
\end{table}

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

\textbf{Note}  
For backward compatibility, wallet properties and SSL APIs are supported; certificate identities established this way are only treated as \texttt{Certificate\#ALL}.

Using a Vault

When running clients, the security subsystem has the opportunity to interact with users to acquire credentials for authentication. This is done using a callback handler as defined by JAAS. However, when running servers (your Visibroker server or a Partition), it is not desirable or even possible to have user interaction at startup time. A typical example of this is if the server is started as a service at the startup time of a host or from an automated script of some sort.

The vault was designed to provide the 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:

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

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

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

*<config <config.jaas-file>>* points to the location of the `config.jaas` file containing the realms 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 want `vaultgen` to execute. You can select any one of the following:

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

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

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

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

```
c:/BDP/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:

```
c:\BDP\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:

```
c:\BDP\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/
```
Client identification

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

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

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

The default behavior of the process can be changed through properties. You can set the retry count by setting `vbroker.security.authentication.retryCount`. The default is 3. The security properties including those for authentication are listed and described in the Chapter 9, “Security Properties for Java” and Chapter 10, “Security Properties for C++.”
Authorization is the process of verifying that the user has the authority to perform the requested operations on the server. For example, when a client accesses an enterprise bean method the application server must verify that the user of the client has the authority to perform such an access. Authorization occurs after authentication (confirming the user's identity).

Authorization is based on the user's identity and an access control list (ACL), which is a list of who can access designated functions. 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 who are then allowed to perform actions in those roles.

Borland uses an access control scheme based on roles. The deployment descriptor maintains a list of roles that are authorized to access each enterprise bean method. The Borland Security Service uses a role database (Role DB) to associate user identities with EJB roles. If a user is associated with at least one allowed role, the user may access the method.

Defining access control with Role DB

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

In VisiBroker, the Role DB file is located with the Security Profiles in the Borland Deployment Platform installation footprint:

<install-dir>/var/security/profiles/<profile-name>/

The default Role DB, default.rolemap, is located in:

<install-dir>/var/security/profiles/default/default.rolemap

In VisiBroker, the location of the rolemap file is specified through the property vbroker.security.domain.<authorization-domain>.rolemap_path
Defining access control with Role DB

The Role DB file is used to determine the access rights of principals (client identities). Each role defined in the Role DB has client identities assigned to it. Access rights are granted based on roles rather than specific client identities. For example, the application may recognize a Sales Clerk role. User identities for all sales clerks can be assigned to the Sales Clerk role. Later, the Sales Clerk role is granted the right to perform certain operations, such as an `add_purchase_order` method, for example. All sales clerks associated with the Sales Clerk role are able to perform `add_purchase_order`.

Anatomy of Role DB

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

```
role-name {
    assertion1 [, assertion2, ... ]
    ...
    [assertion-n]
    ...
}
role-name2 {
    assertion3 [, assertion4, ... ]
    ...
    [assertion-n]
    ...
}
```

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

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

Each rule must contain all necessary security information for a given Principal’s security credentials. That is, each principal must have at least those attributes required from the rule—or exactly all the listed attributes. Otherwise authorization will not succeed.

Assertion syntax

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

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

### Table 4.1 Logical Operators for Authorization Assertions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute = value</td>
<td>equals: attribute must equal value for authorization rule to succeed.</td>
<td>CN=Russ Simmons</td>
</tr>
<tr>
<td>attribute != value</td>
<td>not equal: attribute must not equal value for authorization rule to succeed.</td>
<td>CN!=Rick Farber</td>
</tr>
</tbody>
</table>

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:

```python
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 only look for only one or two attributes from the hierarchy in order to authorize a principal. Borland's security hierarchy starts with GROUPs at the top, then 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:

```python
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:

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

Or, even more simply:

```vbnet
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:

```vbnet
MarketSales {
  role=SalesSupervisor
  *(O=marketing)
}
```

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

### Authorization domains

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" on page 9.

EJBs can be deployed to multiple security contexts with different permissions and roles.

*Note* An authorization domain is associated with an EJB in its deployment descriptor.

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,
- and set up alias(es).
To accomplish these items, the following properties must be set. For more information about these properties, see Chapter 9, “Security Properties for Java” or Chapter 10, “Security Properties for C++”:

### Table 4.2 ORB Properties for Authorization

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbroker.security.authDomains=&lt;domain1&gt;[, &lt;domain2&gt;, &lt;domain3&gt;, ...]</td>
<td>A list of the authorization domain names</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain-name&gt;.defaultAccessRole=grant</td>
<td>deny</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain-name&gt;.rolemap_path=&lt;path&gt;</td>
<td>Path of the Role DB file associated with the authorization domain domain-name. Although this can be a relative path, Borland recommends you make this path fully-qualified.</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain-name&gt;.runas.&lt;role-name&gt;=&lt;alias&gt;</td>
<td>use-caller-identity</td>
</tr>
</tbody>
</table>

### Run-as Alias

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

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.

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

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

### Run-as 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.
CORBA authorization

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

Rolemaps are also used to implement authorization for CORBA objects. Similarly, the J2EE roles and concepts therein are also used in the CORBA environment.

Setting up authorization for CORBA objects

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

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

   **Java**
   ```java
   interface AccessPolicyManager {
       public String domain();
       public com.borland.security.csvc2.ObjectAccessPolicy getAccessPolicy(
           org.omg.PortableServer.Servant servant, byte[] object_id byte[]
           adapter_id);
   }
   ```

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

   This interface should return 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. 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 Borland's implementation. The `ObjectAccessPolicy` interface takes the following form:

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

```cpp
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 Chapter 3, “Authentication.”
Chapter 5: Configuring Security Profiles for Domains

If you have Borland VisiBroker installed, you can define related sets of security parameters for each domain on your system. To enable a profile, you must set certain domain properties. Each profile is configured with the proper security properties, rolemaps, and configuration files. This section explains how to configure security profiles and provide the necessary data for VisiSecure to secure your applications.

Security Profiles

VisiSecure allows you to configure repositories of security information called profiles. Profiles contain the `config.jaas` file for defining LoginModules for authentication, as well as the authentication rolemap, Role DB. Profiles might also contain user databases and script files.

Profiles are defined in your installation’s `security` directory, located in:

```
<install_dir>/var/security/profiles/<profile_name>
```

You can provide a unique name for each of your profiles. The `<profile_name>` directory must contain at least the following files:

- `config.jaas`: the JAAS authentication configuration file, in which your LoginModules are defined.
- `<role_db>.rolemap`: the Role DB file containing authorization data.
- `security.properties`: the central repository for the security properties that help define the operation of the VisiSecure service.
Profiles can also be viewed and configured using the Management Console. To view Security Profiles using the Management Console:

1. From the Hubs View, expand the Management Hubs node.
2. Expand the Agents node.
3. Select the Agent for the domain whose Security Profiles you wish to view.
4. Select the individual Security Profile.
Enabling Security

For a domain to be secure, it must have an enabled Security Profile associated with it. To enable a Security Profile:

1. From the Hubs View, navigate to the profile you want to edit.
2. Right-click the profile and select Configure from the context menu. The Edit Default Properties dialog appears.
3. Check the “Security Enabled” check box.
4. Click OK.

Enabling SSL

To use SSL, your Security Profile must have it enabled. To turn on SSL in a profile:

1. From the Hubs View, navigate to the profile you want to edit.
2. Right-click the profile and select “Configure...” from the context menu. The Edit Default Properties dialog appears.
3. Check the Secure Sockets Enabled check box.
4. Click OK.
Setting the Log Level

To set the Log Level for a Security Profile:

1. From the Hubs View, navigate to the profile you want to edit.
2. Right-click the profile and select Configure from the context menu. The Edit Default Properties dialog appears.

3. Select the desired Logging Level from the Log Level drop-down list.
4. Click OK.

Configuring Authentication

You can configure authentication for your profile by either creating a config.jaas file by hand or by using the Borland Management Console.

Creating the config.jaas file

The config.jaas file contains the data necessary to authenticate a user to one or more realms and defines an authentication mechanism and provides the code to interact with a specific type of authentication mechanism. For example, a config.jaas file could look like this:

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

This defines a realm called UserRealm, which requires the use of the BasicLoginModule. It also provides information on the database used and how to login to it. For information on LoginModules, their options, and the grammar of realm entries such as this, see Chapter 3, “Authentication.”

Once the config.jaas file is complete, it is placed in the profile’s folder.
Configuring Authentication Using the Management Console

To configure authentication using the Management Console:

1. From the Hubs View, navigate to the profile you want to edit.

2. Right-click the profile and select Configure from the context menu. The Edit Default Properties dialog appears.

3. Click Authentication. The Authentication Settings dialog appears.

4. The editing window in the Authentication Settings dialog shows the contents of the profile’s `config.jaas` file. Edit or add realm entries. For more information, see Chapter 3, “Authentication.”

5. When you are finished, click OK.
**Configuring Authorization**

You can configure authorization by either creating your own Authorization Rolemap by hand or by using the Management Console.

**About the rolemap file**

The authorization rolemap is captured in a .rolemap file. Typically, you would name this file after your authorization domain (for example, a profile called “default” would typically call its rolemap default.rolemap) but this is not required. The rolemap file, also called *Role DB*, is a map of users to roles, or *access control list*. Typically, an access control list specifies a set of roles that can use a particular resource. The rolemap designates the set of people whose attributes match those of particular roles, and who are then allowed to perform those roles.

VisiSecure provides a mechanism for specifying role names and a set of attributes which define the roll. For example, the contents of Role DB could be:

```plaintext
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. For information on how to define roles and write a customer rolemap, see Chapter 4, “Authorization.”

Once the rolemap file is complete, it is placed in the profile's folder with the config.jaas file.

**Configuring Authorization Using the Management Console**

You use the Authorization Settings dialog to configure Authorization for a Security Profile. With this you can:

- View authorization rolemaps and rules.
- Add, edit, and remove authorization rolemaps for a domain.
- Add, edit, and remove roles within an authorization rolemap.
- Add, edit, and remove rules within a role.
To access the Authorization Settings dialog:

1. From the Hubs View, navigate to the profile you want to edit.

2. Right-click the profile and select Configure from the context menu. The Edit Default Properties dialog appears.

The left pane of the Authorization Settings dialog presents a tree view of the roles and their associated rules within the selected authorization domain. Selecting a node in the tree displays information about that node. There are three levels of nodes:

- **domain**: this node level shows the name of the rolemap file corresponding to the authorization domain. Its child nodes are roles. Selecting this node displays the entire rolemap file.

- **role**: this node represents a particular role within the domain. Selecting this node displays its role entry within the Role DB. Its child nodes are rules.

- **rule**: this node represents an access rule for its parent role. Selecting this node displays the rule entry within the corresponding role entry.
Working with Authorization Rolemaps and Domains
To edit the authorization rolemap:
1. Open the Authorization Settings dialog.
2. Select the domain node in the left-hand pane. The authorization rolemap file appears.
3. Edit the file in the content window.
4. When you are finished, Click OK.

To add a new authorization rolemap:
1. Open the Authorization Settings Dialog.
2. Click the Add button. The Add Domain dialog appears.
3. Enter a name for the new domain to which the new rolemap will belong and click OK.
4. The new rolemap is presented in the Authorization Settings dialog. You can switch to other rolemaps using the drop-down list at the top of the window.
5. Add roles and rules as necessary.
6. Click OK when you are finished.

To clone an existing rolemap:
1. Open the Authorization Settings Dialog.
2. Click Clone. The Clone Domain dialog appears.
3. Enter a name for the new domain to which the cloned rolemap will belong and click OK.
4. The new rolemap is presented in the Authorization Settings dialog. You can switch to other rolemaps using the drop-down list at the top of the window.
5. Add or edit roles and rules as necessary.
6. Click OK when you are finished.

To remove an authorization domain and its rolemap:
1. Open the Authorization Settings dialog.
2. Select the domain you wish to remove from the drop-down box at the top of the Window.
3. Click the Remove button. The domain is deleted.
4. Click OK when you are finished.
Editing Individual Roles
To edit a rule for an existing authorization role:

1. Open the Authorization Settings dialog.
2. Select the rule node in the left-hand pane. The rule representation appears in the content pane.
3. Click Edit button. The Edit Rule in Role dialog appears.

4. Edit the attribute, conditional operator, and value of the rule in the boxes provided.
5. If you would like to add an additional condition to the rule, click More Conditions. A new row appears.

6. Add additional attributes, operators, and values as required. You can remove the last condition by clicking the Fewer Conditions button.
7. To force strict access based on the rule, check the Match all conditions below exactly and nothing else checkbox.
8. Click preview to display the edited rule.
9. Click OK when you are finished.
Adding and Removing Roles
To add a new role to a rolemap:

1. Open the Authorization Settings dialog.
2. Right click the domain node to which you want to add the role. Select New Role from the context menu. The New Role dialog appears.
3. Give your new role a unique name within the authorization domain and Click OK.
4. The new role appears in the navigation pane. Click OK when you are finished.

To remove an existing role from a rolemap:
1. Open the Authorization Settings dialog.
2. Right click the role node you want to remove. Select Delete Role from the Context menu.
3. The role is removed from the rolemap file. Click OK when you are finished.

Adding and Removing Rules
To add a rule to a role entry:

1. Open the Authorization Settings dialog.
2. Right click the role node to which you want to add the rule. Select New Rule from the context menu. The New Rule in Role dialog appears.

3. Edit the attribute, conditional operator, and value of the rule in the boxes provided.
4 If you would like to add an additional condition to the rule, click More Conditions. A new row appears.

5 Add additional attributes, operators, and values as required. You can remove the last condition by clicking the Fewer Conditions button.

6 To force strict access based on the rule, check the Match all conditions below exactly and nothing else checkbox.

7 Click preview to display the edited rule.

8 Click OK when you are finished. The new rule appears in the tree.

9 Click OK when you are finished.

To remove a rule from a role:
1 Open the Authorization Settings dialog.
2 Right click the rule you want to delete.
3 Select Delete Rule from the context menu. The rule is removed from the tree.
4 Click OK when you are finished.

Specifying VisiSecure properties

Each profile contains a `security.properties` file which allows you to customize the behavior of VisiSecure. A typical properties file could look like the following:

```
# Disable user domain security by default
vbroker.security.disable=false

# Point the ORB at the authentication config files
vbroker.security.login=false
vbroker.security.authentication.config=${profile.path}/config.jaas

# Name the supplied authorization domain
vbroker.security.authDomains=default
vbroker.security.authDomains.default=default

# How to handle requests for methods not in the rolemap file - (grant|deny)
vbroker.security.domain.default.defaultAccessRule=grant
vbroker.security.domain.default.rolemap_path=${profile.path}/default.rolemap
```
Depending on whether your application is Java, C++, or both, you may have to set different properties with different types of values. See Chapter 10, “Security Properties for C++” and Chapter 9, “Security Properties for Java” for all the properties you can set in this file.

Once all the properties have been set, the `security.properties` file is placed in the profile folder.

**Associating a Profile with a Domain**

Security profiles are associated with the various domains on your system by setting properties. Once these properties are set, VisiSecure uses the settings found in the associated security profiles to secure your domains. Each domain on your system has an `orb.properties` file associated with it. This file is located in:

```
<install-dir>/var/domains/<domain_name>/adm/properties/orb.properties
```

To associate a profile and its settings with a domain:

1. Open the domain's `orb.properties` file.
2. Set the following property `vbroker.security.profile` to the name of the profile whose settings you want to use for the domain. For example:

   ```
vbroker.security.profile=default
   ```

VisiSecure will now refer to the settings for the chosen security profile when performing security operations for that domain.

**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. Simply 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 to the profile that can be shared by multiple applications. Otherwise, use the appropriate process-specific ORB properties file to specify the property.
Making Secure Connections (Java)

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

JAAS and JSSE

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

For information on the terms JAAS uses for its services, see “JAAS basic concepts” on page 25.

JSSE Basic Concepts

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

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

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

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

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

**Important**
All security information, including RoleDBs, LoginModule configurations, and such can be set through the Management Console on the appropriate properties tabs.

### 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.

**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.

Keeping these constraints in mind, the Borland VisiBroker 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” on page 66
- “Username/password authentication, using APIs” on page 67
- “Certificate-based authentication, using KeyStores through property settings” on page 67
- “Certificate-based authentication, using KeyStores through APIs” on page 67
- “Certificate-based authentication, using APIs” on page 67
- “pkcs12-based authentication, using KeyStores” on page 67
- “pkcs12-based authentication, using APIs” on page 67

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

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

```
vbroker.security.login=true
vbroker.security.login.realms=known-realm
```
Username/password authentication, using APIs
The following code sample demonstrates the use of the login APIs. This case uses a wallet. For a full description of the four login modes supported, go to the VisiSecure for Java API and SPI sections.

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

Certificate-based authentication, using KeyStores through property settings
By setting the property `vbroker.security.login.realms=Certificate#ALL`, the client will be prompted for keystore location and access information. For valid values, see “Certificate mechanism” on page 38.

Certificate-based authentication, using KeyStores through APIs
You can use the same APIs discussed in “Username/password authentication, using APIs” on page 67 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.

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

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

pkcs12-based authentication, using KeyStores
You can use the same APIs discussed in “Username/password authentication, using APIs” on page 67 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
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 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 to set target trust (whether or not targets must authenticate), the transport policy (whether or not to use SSL or another secure transport mechanism specified separately), 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 use javax.net.ssl.trustStore property to set it. A default X509TrustManager provided by the security service will be used if one is not provided.

Other trust policies are set in the QoP configurations. See “Step Two: Setting properties and Quality of Protection (QoP)” on page 68.

Step Four: Setting up the Pseudo-Random Number Generator

Setting up the PRNG is required if you intend to use SSL communication. Construct a SecureRandom object and seed it. Set this object as your PRNG by using the com.borland.security.Context interface, setSecureRandom method. For detailed information on the com.borland.security.Context interface, go to the VisiSecure for Java API and SPI book.

Step Five: If necessary, set up identity assertion

When a client invokes a method in a mid-tier server which, in the context of this request, invokes an end-tier server, then the identity of the client is internally asserted by the mid-tier server by default. Therefore, if getCallerPrincipal is called on the end-tier server, it will return the Client's principal. Here the client's identity is asserted by the mid-tier server. The identity can be a username or 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, go to the VisiSecure for Java API and SPI book.
Examining SSL related information

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

Clients To examine peer certificates, use `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.

Creating Custom Plugins

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

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

LoginModules

You can implement your own LoginModules by extending `javax.security.auth.spi.LoginModules`. To use the LoginModule, you need to set it in the authentication configuration file, just like any other LoginModule. During runtime, the new customized module will need to be loaded by the secured application.

The syntax of the authentication configuration is as follows:

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

CallbackHandlers

You can implement your own callback by extending `javax.security.auth.callback.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. During runtime, the new customized module will need to be loaded by the secured application.

Authorization Service Provider

Authorization is the process of making access control decisions on behalf of certain resources based on security attributes or privileges. VisiSecure uses the notion of Permission in authorization. The class `RolePermission` is defined to represent a “role” as a permission. Authorization Services Providers in turn provide the implementation on the homogeneous collection of role permissions that associate privileges with particular resources.
Authorization service providers are tightly connected with Authorization Domains. Each domain has exactly one authorization service provider implementation. During the initialization of the ORB, the authorization domains defined by `vbroker.security.authDomains` is constructed, while the Authorization Service Provider implementation is instantiated during the construction of domain itself.

To plugin authorization service, you need to set properties:

```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.

**Trust Providers**

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

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

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

The properties specified will be passed to the user plugin following the same mechanism as above.
Making Secure Connections (C++)

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

Steps to secure clients and servers

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

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

Important
All security information, including RoleDBs and LoginModule configurations, can be set through the Management Console on the appropriate properties tabs.

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.

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 certificate chain.
Steps to secure clients and servers

Keeping these constraints in mind, Borland 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 LoginModules, for known realms” on page 72
- “Username/password authentication, using APIs” on page 72
- “Certificate-based authentication, using KeyStores through property settings” on page 72
- “Certificate-based authentication, using KeyStores through APIs” on page 72
- “Certificate-based authentication, using APIs” on page 73
- “pkcs12-based authentication, using KeyStores” on page 73
- “pkcs12-based authentication, using APIs” on page 73

**Username/password authentication, using LoginModules, for known realms**

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

```
visbroker.security.login=true
visbroker.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 Chapter 11, “VisiSecure for C++ APIs” and Chapter 12, “Security SPI for C++.”

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

**Certificate-based authentication, using KeyStores through property settings**

By setting the property `visbroker.security.login.realms=Certificate#ALL`, the client will be prompted for keystore location and access information. For valid values, see “Certificate mechanism” on page 38.

**Certificate-based authentication, using KeyStores through APIs**

You can use the same APIs discussed in “Username/password authentication, using APIs” on page 72 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.
Steps to secure clients and servers

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

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

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

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

**pkcs12-based authentication, using KeyStores**
You can use the same APIs discussed in "Username/password authentication, using APIs" on page 72 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.

**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 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)” on page 73.
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.

Examining SSL related information

Borland VisiBroker provides APIs to inspect and set SSL-related information. The SecureContext API is used to inspect the SSL ciphersuites and enable select ciphers.

Clients

To examine peer certificates, use getPeerSession() to return an SSLSession object associated with the target. You can then use 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.

Creating Custom Plugins

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

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

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

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

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

You can implement your own LoginModules by extending `vbsec::LoginModule`. To use the LoginModule, you need to set it in the authentication configuration file, just like any other LoginModule. During runtime, the new customized module will need to be loaded by the secured application.

The syntax of the authentication configuration is as follows:

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

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

CallbackHandlers

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

Authorization Service Provider

Authorization is the process of making access control decisions on behalf of certain resources based on security attributes or privileges. VisiSecure uses the notion of Permission in authorization. The class `RolePermission` is defined to represent a “role” as a permission. Authorization Services Providers in turn provide the implementation on the homogeneous collection of role permissions that associate privileges with particular resources.

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

To plugin authorization service, you need to set 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.
Trust Providers

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

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

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

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

The Borland VisiBroker Server allows you to secure the web components using the conventions of encryption, authentication, and authorization.

Like the security measures provided by the J2EE standards, where security is set at the module level, you can also secure web components by declaring the security mechanism within their configuration files.

Security for the Apache web server

The Apache web server uses data transport encryption technology for security. VisiBroker supports the mod_ssl module for this purpose. This module provides strong cryptography for Apache Web Server via the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols by the help of the Open Source SSL/TLS toolkit OpenSSL.

The VisiBroker Server provides mod_ssl which is now supported directly into the Apache web server, and is based on OpenSSL version 0.9.6g.

Modifying the Apache configuration file for mod_ssl

To enable mod_ssl, you must modify the httpd.conf file located in: <install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apacheManagedObject_name>/conf by un-commenting the following line:

```
LoadModule ssl_module <install_dir>/lib/<apacheManagedObject_name>/mod_ssl.so
```
The following describes the mod_ssl directives:

```<IfModule mod_ssl.c>
 SSLEngine on
 SSLRandomSeed startup builtin
 SSLRandomSeed connect builtin
 SSLCertificateFile <install_dir>/var/domains/<domain_name>/configurations/
 <configuration_name>/mos/<apacheManagedObject_name>/conf/ssl.crt/server.crt
 SSLCertificateKeyFile <install_dir>/var/domains/<domain_name>/configurations/
 <configuration_name>/mos/<apacheManagedObject_name>/conf/ssl.key/server.key
 # Uncomment the following to enable SSL certificate and related information to
 # be exported to the Apache environment
 #SSLOptions +StdEnvVars +ExportCertData
</IfModule>
```

**Warning** The mod-ssl Apache module causes the Apache web server to become unstable when used with the KeepAlive connection option on (the default). This is a known defect and users should use caution after enabling the mod-ssl Apache module.

**Table 8.1 mod_ssl directives**

<table>
<thead>
<tr>
<th>SSL-specific directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSLEngine</td>
<td>The placement of the SSLEngine is significant. It can be placed either at the server level, in which case the server will respond only to HTTPS connections or within a particular virtual host, which can then be associated with a particular port number (usually 443), so that both regular HTTP connections and HTTPS connections can be handled.</td>
</tr>
<tr>
<td>SSLRandomSeed</td>
<td>Determines the source of randomness used by the mod_ssl encryption facilities. The randomness built into mod_ssl is sufficient to get you started, however, it is not really random enough to be used in a truly secure environment. Preferably, a UNIX random device such as /dev/random or /dev/urandom is used. The SSLRandomSeed directive must be defined at the server level.</td>
</tr>
<tr>
<td>SSLCertificateFile</td>
<td>The placement of the SSLCertificateFile is significant. It can be placed either at the server level, in which case the server will respond only to HTTPS connections or within a particular virtual host, which can then be associated with a particular port number (usually 443), so that both regular HTTP connections and HTTPS connections can be handled. This file can be given any name and may be placed in any accessible directory.</td>
</tr>
<tr>
<td>SSLCertificateKeyFile</td>
<td>The placement of the SSLCertificateKeyFile is significant. It can be placed either at the server level, in which case the server will respond only to HTTPS connections or within a particular virtual host, which can then be associated with a particular port number (usually 443), so that both regular HTTP connections and HTTPS connections can be handled. This file can be given any name and may be placed in any accessible directory.</td>
</tr>
<tr>
<td>SSLOptions +StdEnvVars +ExportCertData</td>
<td>By default, commented out. If you want to enable Certificate Passthrough, you must uncomment this directive which instructs mod_ssl to export the SSL certificate and related information passed to it by the browser into a shared environment.</td>
</tr>
</tbody>
</table>

**Important** The VisiBroker Server does not provide the key_file and certificate_file which must be generated. See “Creating key and certificate files” on page 79.

For additional information on mod_ssl configuration, visit [http://www.modssl.org/docs](http://www.modssl.org/docs).
Creating key and certificate files

The VisiBroker Server provides the “openssl” utility so that you can generate the required key and certificate files for mod_ssl. The openssl utility is located in:

\(<install_dir>/bin/<apache_ManagedObject_name>/openssl/\)

- For Windows, after double-clicking the openssl executable, a command window appears.
- For UNIX, simply follow the steps below.

**Note** UNIX also requires a random data source for seeding. If you do not have a /dev/rand device installed, you need to provide a file with a random number greater than 512 bytes in length.

The openssl executable first searches the environment for a variable named “RANDFILE”. If that is found, that value is assumed to be a file containing at least 512 bytes of data. If the environment variable RANDFILE is not found, the executable searches the root of your home directory for a <file_name>.rnd. If that is found, it is assumed to contain at least 512 bytes of data for the seed. If you do not have a /dev/rand device, and do not provide any other alternative, certificate generation will fail.

To generate the files:

1. Create a private key for your server:
   
   ```
   OpenSSL> genrsa -out <key_file>
   ```

2. Generate a certificate request:
   
   ```
   OpenSSL> req -new -key <key_file> -out <request_file> -config <install_dir>/bin/<apache_ManagedObject_name>/openssl.cnf
   ```

3. Create a temporary certificate:
   
   ```
   OpenSSL> req -x509 -key <key_file> -in <request_file> -out <certificate_file> -config <install_dir>/bin/<apache_ManagedObject_name>/openssl.cnf
   ```

Using the configuration from:

```
<install_dir>/bin/<apache_ManagedObject_name>/openssl.cnf
```

You are prompted for the following information.

**Note** Pressing Enter in response to each query (accepting each default value) is sufficient for creating a temporary certificate.

```
Using configuration from
<install_dir>/bin/<apache_ManagedObject_name>/openssl.cnf
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank.
For some fields there will be a default value.
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (such as, city) []:
Organization Name (such as, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (such as, section) []:
Common Name (such as, YOUR name) []:
Email Address []:
```
Please enter the following 'extra' attributes to be sent with your certificate request:
A challenge password []:
An optional company name []:
Using configuration from <install_dir>/bin/<apache_ManagedObject_name>/openssl.cnf

4 The key file created must be moved to the location specified in SSLCertificateKeyFile.
5 The certificate file must be moved to the location specified in SSLCertificateFile.

Note: The generated certificate file is sufficient for testing purposes. However, in a production environment, you must obtain a certificate signed by a recognized certificate authority (CA).

Verifying that mod_ssl is active

In order to verify that mod_ssl is working, try to access your web server using https. How you do this depends on where you have placed the SSLEngine directive (see “Creating key and certificate files” on page 79).

- If SSLEngine is defined at the server level, all server access goes through mod_ssl.
- If SSLEngine is defined on the host level, you must direct your access to that particular host by providing the correct port number or IP address.

When attempting to make a secure connection from a web browser, make sure that you use “https” instead of “http” in your URL. For example:

https://host.domain.com:443/index.html

To find out more information about your web server, you can configure mod_info to serve configuration information.

1 To enable mod_info, you must modify the httpd.conf file located in:

<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_ManagedObject_name>/conf

by un-commenting the following line:

LoadModule info_module <install_dir>/lib/<apache_ManagedObject_name>/mod_info.so

2 Next, you need to uncomment the following section of code:

<Location /server-info>
  SetHandler server-info
  Order deny, allow
  # Deny from all
  Allow from .your_domain.com
</Location>

Important: Do not uncomment the Deny from all directive.

3 Edit the Allow from .your_domain.com directive to match your domain.

Your Apache web server will now respond to the following query by displaying server configuration information:

https://<server_name>/server-info

Note: This query does not require a secure “https” connection; mod_info configuration is independent of the protocol used by the Apache web server.
Enabling certificate passthrough to the Borland web container

By enabling mod_ssl for your Apache web server, you can configure your web server to handle “https” (SSL) type connections. As a result, any SSL authentication information is consumed by the Apache web server. If you want your Borland web container to manage the SSL authentication, the Apache web server and IIOP connector need to pass the SSL authentication information through to the Borland web container.

By implementing the VisiBroker “certificate passthrough” feature, you can give your Borland web container access to all the browser-supplied SSL information, as if the Apache web server is not between the browser and the web container. Additionally, your Borland web container is given control of the SSL-based authorization. With this feature, the web applications can use the auth-method CLIENT_CERT_AUTH even when the Apache web server is in between the browser and the Borland web container.

Configuring Apache to “passthrough” the SSL certificate and related information

Enabling certificate passthrough consists of the following two steps:

1. Configuring the mod_ssl module of the httpd.conf file to export the SSL authentication information passed to it by the browser into a shared environment.

   This shared environment allows the mod_iiop IIOP connector to obtain this data for subsequent forwarding to the Borland web container.

2. Configuring the mod_iiop IIOP connector to forward any SSL authentication information to the Borland web container.

Configuring the mod_ssl module of the httpd.conf file for certificate passthrough

Modify the httpd.conf file located in: `<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apacheManagedObject_name>/conf` by uncommenting the `ssloptions +StdEnvVars +ExportCertData` directive in the `mod_ssl` section. For example:

```bash
<IfModule mod_ssl.c>
  BrowserMatch ^Mozilla/[2345] nokeepalive
  SSLEngine on
  SSLRandomSeed startup builtin
  SSLRandomSeed connect builtin
  SSLCertificateFile C:\BDP(b414)\var\domains\base\configurations\j2eeSample/mos/ApacheWebServer/conf/ssl.crt/server.crt
  SSLCertificateKeyFile C:\BDP(b414)\var\domains\base\configurations\j2eeSample/mos/ApacheWebServer/conf/ssl.key/server.key
  # Uncomment the following to enable SSL certificate and related information to be exported to the apache environment
  #SSLOptions +StdEnvVars +ExportCertData
</IfModule>
```

Chapter 8: Security for the Web components 81
Configuring the mod_iioop IIOP connector of the httpd.conf file to forward SSL authentication

As part of the certificate passthrough feature, a new IIOP configuration directive has been added. The `IIopEnableSSLExport` directive instructs the IIOP connector to forward SSL requests.

If the `IIopEnableSSLExport` directive is enabled and a client request is set as secure (has a type of `https`), then the IIOP connector will attempt to locate a set of environment variables. If a set of environment variables are found, then the IIOP connector forwards the data on to the Borland web container as part of the request.

Modify the httpd.conf file located in: `<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apacheManagedObject_name>/conf` by un-commenting the appropriate directives, shown here:

```plaintext
<IfDefine !ModIIOPNoAutoLoad>
  LoadModule iiop2_module C:/BDP(b414)/lib/<apacheManagedObject_name>/mod_iioop2.dll
  IIopLogFile C:\BDP(b414)\var\domains\base\configurations\j2eeSample/mos/
  ApacheWebServer/logs/mod_iioop.log
  IIopLogLevel error
  IIopClusterConfig C:\BDP(b414)\var\domains\base\configurations\j2eeSample/mos/
  ApacheWebServer/conf/WebClusters.properties
  IIopMapFile C:\BDP(b414)\var\domains\base\configurations\j2eeSample/mos/
  ApacheWebServer/conf/UriMapFile.properties

# Directive: IIopLookupLocalRefs true | false
# Purpose: Allows apache to override the UriMapFile mapping if the reference is found in the local doc tree.
# Default: false - by default mod_iioop will not look for local references prior to dispatching a matching reference to
  #   tomcat.
# #IIopLookupLocalRefs true
# #
# Directive: IIopChunkedUploading true | false
# Purpose: Determines whether mod_iioop will attempt to chunk POSTed data (uploads). Corresponds to the enabledChunking attribute of the Borland WebContainer's IIOP connector.
# Default: false - Mod_iioop will not attempt to chunk uploaded data.
# #IIopChunkedUploading true
# #
# Directive: IIopUploadBufferSize n
# Purpose: Allows mod_iioop to control the size of the 'chunk' of data POSTed to the Borland WebContainer as part of a chunked upload. This directive is ignored if IIopNoChunkedUploading is set to true.
# Default: 4096
# #IIopUploadBufferSize 4096
# #
# Directive: IIopReapIdleConnections n
# Purpose: Directs the VisiBroker orb to attempt to reap 'n' idle connections from the orb connection pool after every HTTP request. This can improve performance on systems that have limited network resources.
```
Security for the Borland web container

In order to protect access to your web resources, you must secure those resources. The following steps are required for securing web resources using VisiBroker:

1. Set up security for your Borland web container
2. Set up security for each web application

Securing your Borland web container

By default, the Borland web container is set up to use the Borland Security Service Realm (BSSRealm). To secure your Borland web container, you need to complete the following:

1. Enable security
2. Configure Security authentication
3. Configure Security authorization

Securing your web application

The VisiBroker Server allows you to set security for each of your web applications by protecting the URLs with which the application's resources are associated. To secure a web application, you must first decide which URLs you want to protect. Once you protect a URL, a user is not able to access it without entering a valid user name and password.

Once you identify a web resource collection (servlets, JSPs, HTMLs, Gifs, and such) and the associated URLs you want to protect, the steps to securing the web application are:

1. Define new security roles: Assign users to a security role which is used to determine who accesses the web resources and what actions are allowed when using the web resources by way of a web browser.
2. Define the security constraints for the specific web resource files: Protect the URL Patterns that map to certain servlets and JSPs.
3. Set up a login: Set the login option which controls the access to the servlets and JSPs through their URL patterns.

For example, you may set up a “Developer” security role that can access the web resources of the example.war file which contains the URL Pattern, jsp/security/servlet/*.jsp.

# Default: none - by default mod_iio2 does not request the orb to reap idle connections.
#
#IIopReapIdleConnections 50

# uncomment the following to enable the "certificate passthrough" feature of the IIOP connector
# Note: you must have already un-commented the SSLOptions directive of the mod_ssl module earlier

#IIopEnableSSLExport true

</IfDefine>
Three-tier authorization scheme

In addition to setting up security from your client browser to your Borland web container, you can set up a three-tier authorization scheme to accommodate a more complex client/server landscape. A three-tier authorization scheme can include a client browser, a web container, and an EJB container.

Figure 8.1  Three-tier authorization scheme

The server-side has two different container components with a security mechanism in each of them. So, when a user (John) sends a client request, his login ID is authorized and authenticated at the Borland web container level.

Let’s say that the client request requires the servlet running at the Borland web container to access a bean in the EJB container. However, the EJB container does not know the user, “John”. You have two options for extending security to the EJB container.

- The first is to make the EJB container knowledgeable of all users.
- The second is to use the concept of “run-as”—When the web container makes an EJB invocation, the web container will “run as” a user that the EJB container recognizes. The web application can be configured with a “run as” user to access the third-tier component. The web application with the servlet making the EJB invocation can be configured with “run as” user “web container”. In this case, though the real user is “John”, the EJB container acknowledges the user as “web-container”.

Setting up “run-as” role

The Borland web container supports the “run-as” configuration for web applications. The web application can be set up with the “run-as” role which maps to a user.

To set up the “run as” configuration:
1. Open the .war file in the DDEditor.
2. In the Navigation pane, expand the .war file.
3. Select a servlet node.
4. Access the Properties pane, General tab.
5 In the Security Identity field, choose Run as from the drop-down menu.

6 In the Run section that appears, use the Descriptions field to type a short description of the Run as.

7 In the Role field, use the pull-down menu to select the appropriate role from the list.

Note In addition to setting the “run-as” role, you also must set a principal to map to the “run as” role. For more information, see “Authorization domains” on page 46.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbroker.security.logLevel</td>
<td>Use this property to control the degree of logging. 0 means no logging and 7 means maximum logging (debug messages).</td>
<td>0</td>
</tr>
<tr>
<td>vbroker.security.secureTransport</td>
<td>This property controls whether the transport connection is encrypted or not. If set to true, transport messages are encrypted. If set to false they are in the clear.</td>
<td>true</td>
</tr>
<tr>
<td>vbroker.security.alwaysSecure</td>
<td>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.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.server.transport</td>
<td>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.</td>
<td>SECURE_ONLY</td>
</tr>
<tr>
<td>vbroker.security.disable</td>
<td>If set to true, disables all security services.</td>
<td>true</td>
</tr>
<tr>
<td>vbroker.security.transport.protocol</td>
<td>This property is used to select a security transport protocol. Possible values are SSL, SSLv2, SSLv3, TLS and TLSv1. For information on these protocols, see the Sun Microsystems documentation at: <a href="http://java.sun.com/products/jse/doc/guide/API_users_guide.html#SSC">http://java.sun.com/products/jse/doc/guide/API_users_guide.html#SSC</a>.</td>
<td>TLSv1</td>
</tr>
<tr>
<td>vbroker.security.requireAuthentication</td>
<td>Server-side only property used to specify whether the client is required to authenticate.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.enableAuthentication</td>
<td>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.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.authentication.callbackHandler</td>
<td>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.</td>
<td>n/a</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>vbroker.security.authentication.config</td>
<td>This specifies the path to the configuration file used for authentication.</td>
<td>null</td>
</tr>
<tr>
<td>vbroker.security.authentication.retryCount</td>
<td>Number of times to retry if remote authentication failed.</td>
<td>3</td>
</tr>
<tr>
<td>vbroker.security.authentication.clearCredentialsOnFailure</td>
<td>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.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.login</td>
<td>If set to true, at initialization-time this property tries to login to all the realms listed by property vbroker.security.login.realms.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.login.realms</td>
<td>This gives a list of comma-separated realms to login to. This is used when login takes place, either through property vbroker.security.login(set to true) or API login using login().</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.vault</td>
<td>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.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.identity.reauthenticateOnFailure</td>
<td>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.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.identity.enableReactiveLogin</td>
<td>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.</td>
<td>true</td>
</tr>
<tr>
<td>vbroker.security.authDomains</td>
<td>Specifies a comma-separated list of available authorization domains. For example: vbroker.security.authDomains=&lt;dom1&gt;,&lt;dom2&gt;...</td>
<td>null</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain_name&gt;.rolemap_path</td>
<td>Specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain &lt;domain_name&gt; specified in vbroker.security.authDomains.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain_name&gt;.rolemap_enableRefresh</td>
<td>When set to true, enables dynamic loading of the RoleDB file specified in vbroker.security.domain.&lt;domain_name&gt;.rolemap_path property. The interval of dynamic loading is specified by property vbroker.security.domain.&lt;domain_name&gt;.rolemap_refreshTimeInSeconds.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain_name&gt;.rolemap_refreshTimeInSeconds</td>
<td>Specifies the rolemap refresh time in seconds.</td>
<td>300</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain_name&gt;.runas.&lt;run_as_role_name&gt;</td>
<td>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.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.peerAuthenticationMode</td>
<td>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.</td>
<td>NONE</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>vbroker.security.trustpointsRepository</td>
<td>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:&lt;path_to_certs&gt; or a Keystore, given in the format Keystore:&lt;path_to_keystore&gt;.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.default.JSSETrust</td>
<td>If set to true, the JSSE default trust files like cacerts and jssecerts, if present in JRE, will be used to load trusted certificates.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.assertions.trust.&lt;n&gt;</td>
<td>This property is used to specify a list of trusted roles (specified with the format &lt;role&gt;@&lt;authorization_domain&gt;). &lt;n&gt; 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.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.assertions.trust.all</td>
<td>Setting to true will trust all the assertion made by peers.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.cipherList</td>
<td>Set this to a list of comma-separated ciphers to be enabled by default on startup. If not set, a default list of ciphersuites will be enabled. These should be valid SSL Ciphers.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.controlAdminAccess</td>
<td>Set this to true for enabling Server Manager operations on a Secure Server.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.serverManager.authDomain</td>
<td>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.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.serverManager.role.all</td>
<td>Specifies the role name required for accessing all Server Manager operations.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.serverManager.role.&lt;method_name&gt;</td>
<td>Specifies the role name required for accessing the specified method of the Server Manager.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.support.gatekeeper.replyForSAS</td>
<td>This property is used with GateKeeper with security enabled. When set to true, the username and password will not be delegated to the backend server for authentication.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.domain.&lt;domain_name&gt;.defaultAccessRule</td>
<td>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.</td>
<td>grant</td>
</tr>
<tr>
<td>vbroker.se.iiop_tp.scm.ssl.listener.trustInClient</td>
<td>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.default.JSSETrust property.</td>
<td>false</td>
</tr>
<tr>
<td>vbroker.security.wallet.type</td>
<td>A wallet is a set of directories containing encrypted private keys and certificate chains for each identity. Use this property to point to the directory containing the directories for all identities, using the format Directory:&lt;path_to_identities&gt;.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.wallet.identity</td>
<td>Use to point to a directory within the path defined in vbroker.security.wallet.type that contains keys and/or certificate information for a specific identity. Note that the value of this property must consist only of lower-case letters.</td>
<td>n/a</td>
</tr>
<tr>
<td>vbroker.security.wallet.password</td>
<td>Specifies the password used to decrypt the private key or the password associated with the login.</td>
<td>n/a</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>vbroker.security.logLevel</td>
<td>Use this property to control the degree of logging. Acceptable values are: LEVEL_WARN, LEVEL_NOTICE, LEVEL_INFO, and LEVEL_DEBUG strings.</td>
<td>LEVEL_WARN</td>
</tr>
<tr>
<td>vbroker.security.logFile</td>
<td>Use this property to redirect the log output to a file. The default log output is to <code>std::cerr</code>.</td>
<td><code>null</code></td>
</tr>
<tr>
<td>vbroker.security.secureTransport</td>
<td>This property sets whether secure transport is supported or not. If set to <code>false</code>, transport uses CLEAR_ONLY.</td>
<td><code>true</code></td>
</tr>
<tr>
<td>vbroker.security.alwaysSecure</td>
<td>This is a client side only property. It determines whether to use secure transport only or not. <strong>Note:</strong> To use secure transport only, the <code>secureTransport</code> property must also be set to <code>true</code>.</td>
<td><code>true</code></td>
</tr>
<tr>
<td>vbroker.security.server.transport</td>
<td>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 <code>secureTransport</code> property is set to <code>false</code>.</td>
<td>SECURE_ONLY</td>
</tr>
<tr>
<td>vbroker.security.disable</td>
<td>If set to <code>true</code>, disables all security services.</td>
<td><code>false</code></td>
</tr>
<tr>
<td>vbroker.security.requireAuthentication</td>
<td>A server side only property. Use to specify whether the client is required to authenticate.</td>
<td><code>true</code></td>
</tr>
<tr>
<td>vbroker.security.authentication.callbackHandler</td>
<td>Specifies the callback handler for login modules to use for interacting with the user for credentials. You can specify one of the following or your own custom callback handler. For more information, see Chapter 11, “VisiSecure for C++ APIs.”</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HostCallbackHandler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>com.borland.security.provider.authn.CmdLineCallbackHandler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>com.borland.security.provider.authn.HostCallbackHandler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnlineCallbackHandler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnlineCallbackHandler has password echo on, while HostCallbackHandler has password echo off.</td>
</tr>
<tr>
<td>vbroker.security.authentication.config</td>
<td>This specifies the path to the configuration file used for authentication.</td>
<td><code>null</code></td>
</tr>
<tr>
<td>vbroker.security.authentication.retryCount</td>
<td>Number of times to retry if remote authentication failed.</td>
<td><code>3</code></td>
</tr>
<tr>
<td>vbroker.security.login</td>
<td>If set to <code>true</code>, at initialization-time this property tries to login to all the realms listed by property <code>vbroker.security.login.realms</code>.</td>
<td><code>true</code></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>vbroker.security.login.realms</code></td>
<td>This gives a list of comma-separated realms to login to. This is used when login takes place, either through property <code>vbroker.security.login</code> (set to <code>true</code>) or API login.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.vault</code></td>
<td>This property is used to specify the path to the vault file. This property will take effect regardless of whether <code>vbroker.security.login</code> is set to <code>true</code> or <code>false</code>.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.identity.reactiveLogin</code></td>
<td>When set to <code>true</code>, the security service behaves as follows. If the security service cannot find an identity for any of the targets supported by a server it is attempting to communicate with, it then attempts to acquire credentials for one of the targets in the target object's IOR. If a corresponding authentication realm is available for this target (that the user chooses to provide credentials for), then authentication is also attempted locally. Reactive login requires a callback handler to be set either using the appropriate property or at runtime by calling the appropriate method. The default handler is <code>HostCallbackHandler</code>.</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>vbroker.security.authorizations</code></td>
<td>Specifies a comma-separated list of available authorization domains. For example: <code>vbroker.security.authorizations=domain1, domain2</code>.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.domain.&lt;domain-name&gt;.rolemap_path</code></td>
<td>Specifies the location of the RoleDB file that describes the roles used for authorization. This is scoped within the domain specified in: <code>vbroker.security.authorizations</code>.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.domain.&lt;domain-name&gt;.rolemap_enableRefresh</code></td>
<td>When set to <code>true</code>, enables dynamic loading of the RoleDB file specified in <code>vbroker.security.domain.&lt;domain-name&gt;.rolemap_path</code> property. The interval of dynamic loading is specified by property <code>vbroker.security.domain.&lt;domain-name&gt;.rolemap_refreshTimeInSeconds</code>.</td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>vbroker.security.domain.&lt;domain-name&gt;.rolemap_refreshTimeInSeconds</code></td>
<td>Specifies the rolemap refresh time in seconds.</td>
<td><code>300</code></td>
</tr>
</tbody>
</table>
| `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; if not valid, the connection will be refused. If required, transport identity can be established using these certificates. In this mode, peer certificates are not required to be trusted.  
  
  **REQUIRE AND TRUST**—Same as REQUIRE mode, except that the peer certificates need to be trusted, otherwise the connection will be refused.  
  
  **REQUEST**—Peer certificates will be requested. The peer is not required to have certificates; no transport identity will be established when peer does not have certificates. However, if a peer does present certificates, the certificates will be authenticated; if not valid, the connection will be refused. If required, transport identity can be established using these certificates. In this mode, peer certificates are not required to be trusted.  
  
  **REQUEST AND TRUST**—Same as REQUEST mode except that the peer certificates need to be trusted, otherwise the connection will be refused.  
  
  **NONE**—Authentication is not required. During handshake, no certificate request will be sent to the peer. Regardless of whether the peer has certificates, a connection will be accepted. There will be no transport identity for the peer.                                                                                                                                                                                                 | `REQUIRE AND TRUST` |
### Security Properties for C++

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbroker.security.trustpointsRepository</code></td>
<td>Specifies a path to the directory containing trusted certificates. These are given in the form <code>Directory:&lt;certs_dir&gt;</code>. For example: <code>vbroker.security.trustpointsRepository=Directory:c:\data\identities\Delta</code></td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.assertions.trust.&lt;n&gt;</code></td>
<td>Use to specify a list of trusted roles (specify with the format <code>&lt;role&gt;@&lt;authorization_domain&gt;</code>). <code>&lt;n&gt;</code> is uniquely identified for each trust assertion rule as a list of digits. For example, setting <code>vbroker.security.assertions.trust.1=ServerAdmin@default</code> means this process trusts any assertion made by the <code>ServerAdmin</code> role in the <code>default</code> authorization domain.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.assertions.trust.all</code></td>
<td>Setting to <code>true</code> will trust all the assertion made by peers.</td>
<td>false</td>
</tr>
<tr>
<td><code>vbroker.security.server.requireUPIdentity</code></td>
<td>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 <code>true</code>. If <code>vbroker.security.login.realms</code> is set, this property is automatically set to <code>true</code>. However, you can override it by explicitly setting it in the property file.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.cipherList</code></td>
<td>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.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.wallet.type</code></td>
<td>A wallet is a set of directories containing encrypted private keys and certificate chains for each identity. Use this property to point to the directory containing the directories for all identities, using the format: <code>Directory:&lt;path_to_identities&gt;</code></td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.wallet.identity</code></td>
<td>Points to a directory within the path defined in <code>vbroker.security.wallet.type</code> that contains keys and/or certificate information for a specific identity.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.wallet.password</code></td>
<td>Specifies the password used to decrypt the private key or the password associated with the login.</td>
<td>n/a</td>
</tr>
<tr>
<td><code>vbroker.security.CRLRepository</code></td>
<td>Use to specify 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. The CRL file must be in the DER format. 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” on page 16.</td>
<td>n/a</td>
</tr>
</tbody>
</table>
VisiSecure for C++ APIs

This section describes APIs that are defined in VisiSecure for C++. It is separated into subsections including:

- General APIs
- SSL and Certificate APIs
- QoP APIs
- Authorization APIs

All classes are under namespace `vbsec` unless otherwise specified.

General API

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

**class vbsec::Current**

Current represents the view to the thread specific security context. Some calls are relevant only in an request execution context. This object can be obtained through the following code:

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

**Include File**
The `vbsec.h` file should be included when you use this class.

**Methods**

```cpp
void asserting (const vbsec::Subject* caller)
```
Assert a subject as caller identity.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller</td>
<td>The caller name of the subject.</td>
</tr>
</tbody>
</table>

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.

const vbsec::Subject* getCallerSubject ()

Accesses the caller subject.

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer</td>
<td>A peer object retrieved from the bind.</td>
</tr>
</tbody>
</table>

Exceptions

BAD_OPERATION is thrown if this method is called outside the context of a request or when called in a request context where the request was received over a clear TCP connection.

class vbsec::Context

Context represents the security context under which a client will execute. This class can be obtained through the following code:

```
CORBA::Object_var obj = orb->resolve_initial_references("VBSecurityContext");
Context* c = dynamic_cast(obj.in());
```

Include File

The vbsec.h file should be included when you use this class.

Methods

void login()

Login into the system. This logs-in to the realms defined in the property vbroker.security.loginRealms. It traverses the list of realms specified and authenticates against each realm.

void login (vbsec::CallbackHandler& handler)
Use this to login to the system using the specified `CallbackHandler` to obtain the login information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handler</td>
<td>The default callback handler to be used for acquiring information.</td>
</tr>
</tbody>
</table>

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

Login into the system for a specific realm.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm</td>
<td>The realm to login to.</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm</td>
<td>The realm to login to.</td>
</tr>
<tr>
<td>handler</td>
<td>The default callback handler to be use for acquiring information.</td>
</tr>
</tbody>
</table>

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

Login into the system with a `wallet`. `Wallet` can be created using `WalletFactory` API.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wallet</td>
<td>The wallet to be used for login.</td>
</tr>
</tbody>
</table>

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

Login into the system with a set of wallets specified as a vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wallet</td>
<td>A wallet to be used for login.</td>
</tr>
</tbody>
</table>

```cpp
const vbsec::Subject* getSubject (const std::string& realm)
```

Gets the `Subject` corresponding to a given realm.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm</td>
<td>The Realm for the Principal</td>
</tr>
</tbody>
</table>

Returns A pointer to the `Subject` object representing the subject of the realm.

```cpp
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>Stream that the vault information will be read from, in binary format.</td>
</tr>
<tr>
<td>vaultPass</td>
<td>Password used to decrypt the vault information.</td>
</tr>
</tbody>
</table>

```cpp
void logout ()
```

Log the user out from all the realms.

```cpp
void logout (const std::string& realm)
```
Log the user out from a given realm.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm</td>
<td>The realm to logout from.</td>
</tr>
</tbody>
</table>

```c
void setCallbackHandler (vbsec::CallbackHandler* handler)
```

Set the default callback handler programmatically. This is similar to using the property `vbroker.security.authentication.callbackHandler`.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handler</td>
<td>The <code>CallbackHandler</code> to be set.</td>
</tr>
</tbody>
</table>

```c
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 (also used to decrypt the vault). The password may be null. The vault contains all of the system's identities.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>The stream that the vault information will be written into, in binary format.</td>
</tr>
<tr>
<td>password</td>
<td>The password used to encrypt the vault information.</td>
</tr>
</tbody>
</table>

```c
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wallet</td>
<td>The wallet to be used for authentication</td>
</tr>
</tbody>
</table>

```c
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wallet</td>
<td>The wallet corresponding to the identity to be imported.</td>
</tr>
</tbody>
</table>

```c
void setPRNGSeed (const CORBA::OctetSequence& seed)
```

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

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>The seed for the PRNG.</td>
</tr>
</tbody>
</table>

```c
ssl::CipherSuiteInfoList* listAvailableCipherSuites()
```

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

**Returns** List of cipher suits that are available but may not be enabled for use with the SSL layer.

```c
void enableCipherSuites (const ssl::CipherSuiteInfoList& suites)
```
Sets the cipher suites that should be enabled for all SSL sessions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>suites</td>
<td>An IDL-generated CipherSuiteInfoList type.</td>
</tr>
</tbody>
</table>

```cpp
ssl::CipherSuiteInfoList* getEnabledCipherSuites()
```

Gets the set of cipher suites that are currently enabled for all SSL sessions.

Returns Cipher suits that are currently enabled for all SSL sessions.

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

Sets the SSL context. This will allow establishing of an SSL session using the information defined in VBSSLContext. A VBSSLContext can be created using the SecureSocketProvider API.

```cpp
VBSSLContext & getSSLContext()
```

Get the VBSSLContext that is set using the setSSLContext() or return a default VBSSLContext object.

Returns The VBSSLContext that will be used for any SSLSession establishment.

### class vbsec::Principal

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

**Include file**
The vbsec.h file should be included when you use this class.

**Methods**

```cpp
std::string getName() const
```

Returns The name of the Principal.

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

Get the string representation of the Principal.

Returns The string representation of the Principal.

### class vbsec::Credential

Credential represents the information used to authenticate an identity, such as user name and password. This is a virtual class.

**Include File**
The vbsec.h file should be included when you use this class.
**class vbsec::Subject**

Subject represents a grouping of related information for a single entity, such as a person. Such information includes the Subject's identities as well as its security-related attributes (passwords and cryptographic keys, for example).

**Include File**
The `vbsec.h` file should be included when you use this class.

**Methods**

- **Principal::set & getPrincipals()**
  
  Gets the principals in the subject.

  **Returns**
  The set of the principals in the subject. Modifying the content of the set will have no effect on the subject.

  **void clearPrincipals()**

  Clears the principals from the subject. All principals in the subject are removed.

- **Credential::set & getPublicCredentials()**

  Get the public credentials in the subject—public keys for example.

  **Returns**
  The set of the public credential in the subject. Modifying the content of the set will have no effect on the subject.

  **void clearPublicCredentials()**

  Clear the public credentials in the subject. All public credentials in the subject will be destroyed and removed.

- **Credential::set & getPrivateCredentials()**

  Get the private credentials in the subject—private keys for example.

  **Returns**
  The set of the private credential in the subject. Modifying the content of the set will have no effect on the subject.

  **void clearPrivateCredentials()**

  Clear the private credentials in the subject. All private credentials in the subject will be destroyed and removed.

- **Principal::set getPrincipals (const type_info& info) const**

  Gets a set of principals in the subject which have the same runtime type information as provided.

  **Parameter**

  - **info**
    The runtime type information that the returned principals shall have.

  **Returns**

  A set of the principals in the subject which have same runtime information as the given one. Modifying the content of the set will have no effect on the subject.

- **Credential::set getPublicCredentials (const type_info& info) const**

  Get set of public credentials in the subject which have the same runtime type information as provided.

  **Parameter**

  - **info**
    The runtime type information that the returned public credential shall have.
Returns A set of the public credential in the subject which have same runtime information as the given one. Modifying the content of the set will have no effect on the subject.

```cpp
Credential::set getPrivateCredentials (const type_info& info) const
```

Get set of private credentials in the subject which have the same runtime type information as provided.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The runtime type information that the returned private credentials shall have.</td>
</tr>
</tbody>
</table>

Returns A set of the private credentials in the subject which have same runtime information as the given one. Modifying the content of the set will have no effect on the subject.

```cpp
class vbsec::Wallet
```

A `Wallet` is a holder of credentials usually used in login API calls. A `Wallet` can be created using `WalletFactory` APIs and contain multiple types of credentials.

**Include File**

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

**Methods**

```cpp
std::string getTarget () const
```

Get the target to which wallet authenticates.

**Returns**

The string representation of the target information.

```cpp
void populateSubject (Subject& subject)
```

Populate the given subject with necessary credentials or other information for authentication.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject</td>
<td>The subject for the wallet to populate.</td>
</tr>
</tbody>
</table>

```cpp
class vbsec::WalletFactory
```

`WalletFactory` is a factory class to create multiple types of wallets.

**Include File**

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

**Methods**

```cpp
Wallet* createCertificateWallet (const std::string& name, const std::string& password, const std::string& alias, const std::string& keyPassword, short usage)
```
Create a certificate wallet using a C++ keystore. The C++ keystore is similar to the Java keystore but is implemented using a directory structure. When logging in with a wallet created by this API, the certificate chain will be used in the SSL layer.

## Parameter Description

| name       | The directory name of the keystore. |
| password   | The password for the keystore, not used for this release. |
| alias      | The alias to be used in the keystore. |
| keypassword| The password for the private key of the given alias. |
| short usage| The usage of the certificate information, CLIENT, SERVER or ALL. |

Returns Certificate wallet that contains the given information.

```c++
Wallet* createCertificateWallet (const CORBAsec::X509CertList& chain,
                                 const CORBAsec::ASN1Object& privkey,
                                 const CSI::UTF8String& password)
```

Create a certificate wallet using a certificate chain, private key and password.

## Parameter Description

| chain      | The certificate chain to create the wallet. |
| privkey    | The private key of the certificate chain. |
| password   | The password for the private key. |

Returns Certificate wallet that contains the given information.

```c++
Wallet* createIdentityWallet (const std::string& username,
                              const std::string& password,
                              const std::string& realm)
```

Create a identity wallet using a username, password and realm that the wallet to which the wallet authenticates.

## Parameter Description

| username   | The username of the identity. |
| password   | The password for the identity. |
| realm      | The realm to which the wallet authenticates. |

Returns Identity wallet that contains the given information.

```c++
Wallet* createIdentityWallet (const std::string& username,
                              const std::string& password,
                              const std::string& realm,
                              const std::vector<std::string>& groups)
```

Create a identity wallet using a username, password, realm to which the wallet authenticates, and a set of group attributes.

## Parameter Description

| username   | The username of the identity. |
| password   | The password for the identity. |
| realm      | The realm to which the wallet authenticates. |
| groups     | A set of group attributes to which the identity belongs. |

Returns Identity wallet that contains the given information.
SSL API

This section explains the various SSL APIs that interact with VisiSecure's SSL implementation.

**class vbsec::SSLSession**

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

**Include File**
The `vbssp.h` file should be included when you use this class.

**Methods**

```cpp
    time_t getEstablishmentTime() const
    Returns The time when the SSL connection was established.
```

```cpp
    const ssl::CipherSuiteInfo& getNegotiatedCipher() const
    Returns The negotiated cipher from the peer for a given SSL connection.
```

```cpp
    const CORBAsec::X509CertList& getPeerCertificates() const
    Returns Peer certificate chain.
```

```cpp
    const CORBAsec::X509Cert* getTrustpoint() const
    Returns The trust point by which the peer is trusted, or null if not.
```

```cpp
    char* getPeerAddress() const
    Returns Peer IP address in a string with the following format: xxx.xx.xx.xx.
```

```cpp
    CORBA::UShort getPeerPort() const
    Returns The port number of the peer on the connection.
```

```cpp
    void prettyPrint (std::ostream& os) const
    Prints the SSLSession information into the given output stream.
```
class vbsec::VBSSLContext

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

Include File
The vbssp.h file should be included when you use this class.

Methods

const CORBAsec::X509CertList& getCertificates() const
Get the certificate chain representing the identity to be used for the SSL layer.

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

void setCipherSuiteList (const ssl::CipherSuiteInfoList& list)
This method is used to specify the ciphers available for the SSL connections.

const ssl::CipherSuiteInfoList& getCipherSuiteList() const
Return the ciphers that are currently used by the SSL layer.

Returns
The ciphers that are currently used by the SSL layer.

void addTrustedCertificate (const CORBAsec::X509Cert_var& trusted)
Programmatically add trusted certificate into the SSL context.

CORBAsec::X509CertList* getTrustedCertificates() const
Get list of certificates that are trusted.

Returns
List of certificates that are trusted.

class ssl::CipherSuiteInfo

CipherSuiteInfo is a structure containing two fields:

- CORBA::ULong SuiteID
- CORBA::String_var Name

This IDL structure contains two fields which describe ciphers according to the SSL specification. The list of SuiteID values and their names is in the include file, ssl_c.h

Include File
The ssl_c.hh file should be included when you use this class.
class CipherSuiteName

This class provides information about the ciphers used in the Security Service.

Include File

The csstring.h file should be included when you use this class.

Methods

static const char* toString (int tag)

Return a standard representation of a supported SSL cipher.

Returns

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>tag associated with the cipher name.</td>
</tr>
</tbody>
</table>

static const int fromString (char* description)

Give the tag associated to the given cipher description.

Returns

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>The stringified description of the cipher.</td>
</tr>
</tbody>
</table>

class vbsec::SecureSocketProvider

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

Include File

The vbssp.h file should be included when you use this class.

Methods

vbsec::VBSSLContext* createSSLContext (const CORBAsec::X509CertList& chain, const CORBAsec::ASN1Object& privkey, const CSI::UTF8String& password)

This method create a SSL context using the given information. The SSL context can then be passed into vbsec::Context and used to establish an SSL connection.

Returns

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain</td>
<td>The certificate chain.</td>
</tr>
<tr>
<td>privkey</td>
<td>The private key object.</td>
</tr>
<tr>
<td>password</td>
<td>The password for the private key.</td>
</tr>
</tbody>
</table>

void setPRNGSeed (const ssl::Current::PRNGSeed& seed)
Sets a seed for the pseudo-random number generator used by the SSL layer.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>The seed for the PRNG.</td>
</tr>
</tbody>
</table>

```cpp
const ssl::CipherSuiteInfoList& listAvailableCipherSuites() const
```

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

**Returns**

List of cipher suits that are available but may not be enabled for use with the SSL layer.

```cpp
const CertificateFactory& getCertificateFactory() const
```

Gets a certificate Factory.

**Returns**

A `CertificateFactory` object.

### class ssl::Current

The `ssl::Current` lets your client application or server object set its private key and offer its certificate information to its peer. This interface also lets you configure the SSL connection and associate your certificates and private key with an SSL connection.

Be aware that private keys and certificates contain header and trailer lines, which mark the beginning and end of the key or certificate. All of the methods offered by this interface for setting private keys and certificate chains require that these header and trailer lines be present. The parsing rules for these lines is:

- The recognized header line format for certificates is:
  ```plaintext
  -----BEGIN CERTIFICATE-----
  ```
- The recognized header line format for private keys is:
  ```plaintext
  -----BEGIN ENCRYPTED PRIVATE KEY-----
  ```
- All header lines must end with a new line character.
- All trailer lines must be preceded with, and end with, a newline character. PEM-style private keys have two additional header lines that other private keys do not have: Proc-Type and DEK-Info. Both of these lines must be present and they must end with new line characters.

This object can be obtained through the following code:

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

### Include File

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

### Methods

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

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

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer</td>
<td>The peer from which you obtain the negotiated cipher.</td>
</tr>
</tbody>
</table>

**Returns**

A value (tag) representing the cipher used. (Use `CipherSuiteName::toString` to get a String representation.)
SSL API

Exceptions

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

CORBAsec::X509CertList_ptr getPeerCertificateChain(CORBA::Object_ptr peer)

This method obtains the peer's certificate chain. It is usually invoked by a client application to obtain information from a server, but a server can optionally request information from a client.

Parameter  Description
peer        The peer from which you obtain the negotiated cipher.

Returns

A value representing the cipher used. (Use CipherSuiteName::toString to get a String representation.)

Exceptions

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

char* getPeerAddress(CORBA::Object_ptr peer)

Returns a description of the socket parameters used by this connection.

Parameter  Description
peer        The peer from which you obtain the information.

Returns

Peer IP address in a string with the following format: xxx.xx.xx.xx

Exceptions

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

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

Tests if the certificate chain of the peer is trusted—that is, if one certificate of the chain is in the trustpoint.

Parameter  Description
peer        The peer from which you obtain the information.

Returns

true if the chain is trusted, false otherwise.

Exceptions

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

trust::Trustpoints_ptr Trustpoints getTrustpointsObject()

Returns a reference to the trustpoint repository. Use this API to access trustpoints object and set trustpoints.

Returns

A reference to the trustpoint repository, which should be assigned to a _var.

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

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

Parameter  Description
seed       The OctetSequence seed for the PRNG.
This method is used in the client or the server to set the certificate chain and private key that must be used for the SSL connections. This is required for servers and optional for clients. Also look at the `peerAuthenticationMode` property documented in Chapter 10, "Security Properties for C++.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain</td>
<td>The certificate chain.</td>
</tr>
<tr>
<td>privkey</td>
<td>The private key used for the SSL connection.</td>
</tr>
<tr>
<td>password</td>
<td>The password for the private key.</td>
</tr>
</tbody>
</table>

Exceptions

CORBA::BAD_PARAM if the user name or password is null.

```c++
void setCipherSuiteList (const ssl::CipherSuiteInfoList& list)
```

This method is used in the client or the server to specify the ciphers available for the SSL connections.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>A comma-separated list of cipher suites.</td>
</tr>
</tbody>
</table>

```c++
ssl::CipherSuiteInfoList* listAvailableCipherSuites()
```

Returns a list of cipher suites available in VisiSecure. You are responsible for freeing memory.

Returns

A list of cipher suites.

```c++
ssl::CipherSuiteInfoList* getCipherSuiteList()
```

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

Returns

A list of cipher suites.

```c++
void setP12Identity (const CORBASEC::ASNIOBJECT& pks12cert, const char* password)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pks12cert</td>
<td>PKCS#12 formatted data.</td>
</tr>
<tr>
<td>password</td>
<td>The private key password.</td>
</tr>
</tbody>
</table>
Certificate API

This API contains classes and methods for working with certificates.

`class vbsec::CertificateFactory`

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

**Include File**
The `vbssp.h` file should be included when you use this class.

**Methods**

- `CORBAsec::X509CertList* importCertificateChain (const CORBAsec::ASN1ObjectList& certs) const`
  Import the certification chain in the form of `CORBAsec::ASN1ObjectList` into `CORBAsec::X509CertList`, which could be used in `VBSSLContext`.
  
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>certs</td>
<td>ASN1ObjectList representation of the certificate chain.</td>
</tr>
</tbody>
</table>

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

- `CORBAsec::X509CertList* importCertificates (const CORBAsec::ASN1ObjectList& certs) const`
  Import the certification list in the form of `CORBAsec::ASN1ObjectList` into `CORBAsec::X509CertList`. Certificates need not be related to each other. The original order is preserved after importing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>certs</td>
<td>ASN1ObjectList representation of certificate list</td>
</tr>
</tbody>
</table>

  Returns `CORBAsec::X509CertList` representation of the certificate list.

- `CORBAsec::ASN1Object* importPrivateKey (const CORBAsec::ASN1Object& key) const`
  Convert the private key from BASE64 or PEM format to DER format.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>ASN1ObjectList representation of private key object.</td>
</tr>
</tbody>
</table>

  Returns DER format of the private key.

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

  Import the certification chain from a PKCS12 file.
Imports a certificate chain from pkcs12 binary.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pkcs12bytes</td>
<td>ASN1ObjectList representation of pkcs12 binary.</td>
</tr>
<tr>
<td>password</td>
<td>Password for the pkcs12 binary.</td>
</tr>
</tbody>
</table>

**Returns** CORBAsec::X509CertList representation of the certificate chain.

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

Import private key from pkcs12 binary.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pkcs12bytes</td>
<td>ASN1ObjectList representation of pkcs12 binary.</td>
</tr>
<tr>
<td>password</td>
<td>Password for the pkcs12 binary.</td>
</tr>
</tbody>
</table>

**Returns** CORBAsec::ASN1Object representation of the private key object.

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

Print out the certification information into an output stream.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>certificate</td>
<td>certificate to be printed.</td>
</tr>
<tr>
<td>stream</td>
<td>stream to which to output.</td>
</tr>
</tbody>
</table>

**Returns** the CertificateFactory.

```cpp
bool passwordForPrivatekey (const CSI::UTF8String& password,
                            const CORBAsec::ASN1Object& privkey) const
```

Test if the given password can decrypt the given private key object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>password</td>
<td>The password to be tested.</td>
</tr>
<tr>
<td>privkey</td>
<td>The private key object to be decrypted.</td>
</tr>
</tbody>
</table>

**Returns** true if decryption is successful, false if not.
class CORBAsec::X509Cert

This class represents an X509 certificate. When a client application binds to a CORBA object, the client uses this interface to obtain the server’s certificate information. The server can use this interface to obtain the client’s certification information, if the client has a certificate.

Include File
The X509Cert_c.hh file should be included when you use this class.

Methods

char* getSubjectDN()

Returns the subject DN contained in the certificate.

Returns

The subject name is returned in the following format:

CN=<value>, OU=<value>, O=<value>, L=<value>, S=<value>, C=<value>

char* getIssuerDN()

Returns the issuer DN contained in the certificate.

Returns

The subject name is returned in the following format:

CN=<value>, OU=<value>, O=<value>, L=<value>, S=<value>, C=<value>

CORBA::OctetSequence * getSignatureAlgorithm()

Returns the signature algorithm used in the certificate.

Returns

The signature algorithm used in the certificate.

CORBA::OctetSequence * getHash(CORBASEC::HashAlgorithm algorithm)

Returns a hash of the certificate.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>The hash algorithm. The possible values are: CORBAsec::MD5, CORBAsec::MD2 and CORBAsec::SHA1</td>
</tr>
</tbody>
</table>

Returns

A hash of the certificate using the specified algorithm.

CORBAsec::ASN1Object_ptr getDER()

Returns the DER encoded form of this certificate.

Returns

The ASN.1 DER encoded form of this certificate (assign to a _var).

CORBAsec::SerialNumberValue_ptr getSerialNumber()

Retrieves the serial number of the certificate.

Returns

The serial number of the certificate.

CORBAsec::X509CertExtensionList_ptr getExtensions()

Returns all the extensions available in this certificate as a list of X509CertExtension.

Returns

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

CORBA::Boolean isValid (CORBA::ULong out date)
Checks if a certificate date is between the valid start and end dates.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>An out argument that is set to the expiration date of the certificate, using UNIX time format.</td>
</tr>
</tbody>
</table>

Returns true if the certificate is valid, false otherwise.

- **CORBA::ULong startDate()**
  - Gets the date from which a certificate's validity starts.
  - Returns an int representing the number of seconds from midnight, January 1st, 1970.

- **CORBA::ULong endDate()**
  - Gets the expiration date of the certificate.
  - 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.
  - 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 true if the certificate is a trustpoint, returns false otherwise.

**class CORBAsec::X509CertExtension**

This class is an IDL structure that represents an X509 certificate extension, as follows:

```
struct X509CertExtension {
    long seq;
    sequence<long> oid;
    boolean critical;
    sequence/octet value;
};
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>A unique number of the extension in the certificate.</td>
</tr>
<tr>
<td>oid</td>
<td>The oid of the extension.</td>
</tr>
<tr>
<td>value</td>
<td>The value of the extension encoded according to the format specified by the oid.</td>
</tr>
</tbody>
</table>

**Include File**

The X509Cert_c.hh file should be included when you use this class.
QoP API

The following section details the Quality of Protection API provided with VisiSecure.

**class vbsec::ServerConfigImpl**

ServerConfigImpl is the implementation of the csiv2::ServerQoPConfig which is an IDL structure as follows:

```
SERVERConfigImpl (  
  CORBA::Boolean disable,  
  CORBA::Short transport,  
  CORBA::Boolean trustInClient,  
  csiv2::AccessPolicyManager* access_manager,  
  const CORBA::StringSequence& realms = _available,  
  CORBA::Short requiredIdentityType = csiv2::ServerQoPConfig::UP_OR_PK,  
  CORBA::Boolean supportIdentityAssertion = static_cast<CORBA::Boolean>(1)
);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable</td>
<td>Whether or not to disable security.</td>
</tr>
<tr>
<td>transport</td>
<td>The transport mechanism to use. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::CLEAR_ONLY: no secure transport is necessary</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::SECURE_ONLY: only secure connections are permitted</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::ALL: any method of transport is allowed</td>
</tr>
<tr>
<td>trustInClient</td>
<td>Whether or not the target requests the client to authenticate. This value is set on CSIV2 layer.</td>
</tr>
<tr>
<td>access_manager</td>
<td>An access manager for the QoP implementation, an implementation of csiv2::AccessPolicyManager defined by the user. If null, it uses a default value.</td>
</tr>
<tr>
<td>realms</td>
<td>The available realms in which to implement the policy.</td>
</tr>
<tr>
<td>requiredIdentityType</td>
<td>The required identity for the QoP policy implementation. The default value is csiv2::ServerQoPConfig::UP_OR_PK Possible values are:</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::ServerQoPConfig::NO_ID</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::ServerQoPConfig::UP</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::ServerQoPConfig::PK</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::ServerQoPConfig::UP_AND_PK</td>
</tr>
<tr>
<td>supportIdentityAssertion</td>
<td>Whether or not the application supports Identity Assertion.</td>
</tr>
</tbody>
</table>

To define the ServerQoPPolicy, you create this object which defines the various characteristics of the policy.

**Include File**
The CSIV2Policies.h file should be included when you use this class.

**class ServerQoPPolicyImpl**

ServerQoPPolicyImpl is the implementation of the csiv2::ServerQoPPolicy. The ServerQoPPolicyImpl object impacts the QoP behaviour of the server.

**Include File**
The CSIV2Policies.h file should be included when you use this class.

**Methods**

```
ServerQoPPolicyImpl (const csiv2::ServerQoPConfig_var& conf);
```
Constructor of the ServerQoPPolicyImpl object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf</td>
<td>ServerQoPConfig object which contains the designed QoP configuration.</td>
</tr>
</tbody>
</table>

virtual csiv2::ServerQoPConfig_ptr config();

Get the ServerQoPConfigImpl object from the ServerQoPPolicyImpl.

Returns

The ServerQoPConfigImpl object from the ServerQoPPolicyImpl.

class vbsec::ClientConfigImpl

ClientConfigImpl is the implementation of the csiv2::ClientQoPConfig. To define the ClientQoPPolicy, you create this object which defines the various characteristics of the policy.

Include File

The CSIV2Policies.h file should be included when you use this class.

Methods

ClientConfigImpl (const CORBA::Short transport, const CORBA::Boolean trustInTarget)

Constructor of ClientConfigImpl object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td>The transport mechanism to use. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::CLEAR_ONLY: no secure transport is necessary</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::SECURE_ONLY: only secure connections are permitted</td>
</tr>
<tr>
<td></td>
<td>■ csiv2::ALL: any method of transport is allowed</td>
</tr>
<tr>
<td>trustInTarget</td>
<td>Whether or not to require the client to authenticate.</td>
</tr>
</tbody>
</table>

class vbsec::ClientQoPPolicyImpl

ClientQoPPolicyImpl is the implementation of the csiv2::ClientQoPPolicy. The ClientQoPPolicyImpl object impacts the QoP behaviour of the server.

Include File

The CSIV2Policies.h file should be included when you use this class.

Methods

ClientQoPPolicyImpl (const csiv2::ClientQoPConfig_var& conf);

Constructor for ClientQoPPolicyImpl object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf</td>
<td>ClientConfigImpl object to be use for the policy.</td>
</tr>
</tbody>
</table>

virtual csiv2::ClientQoPConfig_ptr config();

Returns the ClientConfigImpl object of this ClientQoPPolicyImpl.

Returns

The ClientConfigImpl object of this ClientQoPPolicyImpl.
Authorization API

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

**class csiv2::AccessPolicyManager**

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

**Include File**

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

**Methods**

```cpp
char* domain()
```

Returns the authorization domain name for the `AccessPolicyManager`.

```cpp
csiv2::ObjectAccessPolicy* getAccessPolicy (PortableServer_ServantBase* servant,
const PortableServer::ObjectId& id,
const CORBA::OctetSequence& adapter_id)
```

Returns the `ObjectAccessPolicy` for the servant with the `objectId` (id) and poa id.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servant</td>
<td>The CORBA servant object.</td>
</tr>
<tr>
<td>id</td>
<td>The id of the servant object.</td>
</tr>
<tr>
<td>adapter_id</td>
<td>The poa id of the servant object.</td>
</tr>
</tbody>
</table>

Returns `ObjectAccessPolicy` of the servant object.
class csiv2::ObjectAccessPolicy

This class represents the access policy from AccessPolicyManager.

Include File
The CSIV2Policies.h file should be included when you use this class.

Methods

CORBA::StringSequence* getRequiredRoles (const char* method)

Returned the list of required roles to access the method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>The method name of interest.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>The method name of interest.</td>
</tr>
</tbody>
</table>

Returns
The run-as role configured to access the method.
Chapter 12
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 use within the Borland Deployment Platform.

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. Namespace must be specified in a normalized form supported by VisiSecure, using either a '.' or '::' separated-string starting from the outer namespace. For example:

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

Thus `MyLoginModule` shall be specified as either `MyNameSpace.MyLoginModule` or `MyNameSpace::MyLoginModule`.

There are six pluggable components:

- **LoginModules**: You can implement their own login models 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 their own callback 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 adaptor which also changes target information, `AuthenticationMechanism` is a pluggable service to authenticate users.
To use these plug-ins, you need to set the `vbroker.security.identity.xxx` properties to define the plug-ins and their properties. For example, an identity adapter or mechanism adapter could specify:

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

while an authentication mechanism would provide:

```plaintext
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:

  ```plaintext
  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:

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

- **Trust provider:** This allows you to plug in an assertion trust mechanism. Assertion can happen in multi-hop scenario, or explicitly called through assertion API. The server can have rules to determine whether the peer is trusted to make the assertion or not. The default implementation uses 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:

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

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

Each provider instance is created by the VisiSecure using a Java reflection API. After the instance has been constructed, the `initialize` method, which must be provided by the implementer, is called passing in a map of options specific for the implementation. The options entries are defined by the implementers of the particular provider. Users specify the options in a property file and the VisiSecure parses the property and passes the options to the corresponding provider. The following table shows the properties for plugging in different provider implementations.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Property to set</th>
<th>Interface to implement</th>
<th>Options Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdentityAdapter</td>
<td>vbroker.security.identity.adapters</td>
<td>vbsec::IdentityAdapter</td>
<td>vbroker.security.identity.adapter.&lt;name&gt;</td>
</tr>
<tr>
<td>AuthenticationMechanism</td>
<td>vbroker.security.identity.mechanisms</td>
<td>vbsec::AuthenticationMechanisms</td>
<td>vbroker.security.identity.mechanism.&lt;name&gt;</td>
</tr>
<tr>
<td>AttributeCodec</td>
<td>vbroker.security.identity.attributeCodecs</td>
<td>vbsec::AttributeCodec</td>
<td>vbroker.security.identity.attributeCodec.&lt;name&gt;</td>
</tr>
<tr>
<td>TrustProvider</td>
<td>vbroker.security.trust.trustProvider</td>
<td>vbsec::TrustProvider</td>
<td>vbroker.security.trust.trustProvider.&lt;name&gt;</td>
</tr>
</tbody>
</table>

In the preceding table:

- The first column lists the provider module names.
- The second column lists the property you set to define each module. Use a comma to separate multiple modules. For example, the following property has two additional IdentityAdapter implementations installed for the ORB:

  `vbroker.security.identity.adapters=ID_ADA1,ID_ADA2`

- The third column gives the interface each implementation must implement. The interface defines a contract between the implementers and the core VisiSecure.
- The final column gives the options prefix for the specific module. The ORB parses the property file and passes the corresponding entries to each of the modules in the `initialize` method as the (Map options) parameter. For example, for the `ID_ADA1 IdentityAdapter` defined in the previous example, all the entries with the `vbroker.security.identity.adapters.ID_ADA1` prefix will be passed to the initial method of `ID_ADA1 IdentityAdapter`.

Providers and exceptions

During the initialization, if anything goes wrong the `initialize` method should throw an instance of `InitializationException`. For certain categories of providers, there can be multiple instances with different implementations co-existing. Each of them is identified by the name within the VisiSecure system, which is passed as the first parameter in the `initialize` method. While for some categories of providers there can be only one instance existing for the whole ORB (such as in the case of the TrustProvider, in this case, the `initialize` method has only one single parameter ?the options map.)
LoginModule serves as the parent of all login modules. User plugin login modules must extend this class. Login modules are configured in the authentication configuration file and called during the login process. Login modules are responsible of authenticating the given subject and associating relevant Principals and Credentials with the subject. It is also responsible for removing and disposing of such security information during logout.

Include File

The vbauthn.h file should be included when you use this class.

Methods

void initialize (Subject* subj=0,
                CallbackHandler *handler=0,
                LoginModule::states* sharedStates=0,
                LoginModule::options* options=0)

This method initializes the login module.

Arguments

This method utilizes the following four arguments:

- subj: the subject to be authenticated.
- handler: the callback handler to use.
- sharedStates: 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 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 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

The vbauthn.h file should be included when you use this class.

Methods

void handle (Callback::array& callbacks)
Handle the callbacks.

Arguments  
the array of callbacks to be processed.

Returns  
Void.

vbsec::IdentityAdapter

IdentityAdapter binds to a particular mechanism. The main purpose of an IdentityAdapter is to interpret identities 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 with the VisiSecure:
- AnonymousAdapter, with the name "anonymous"
- DNAdapter, with the name "DN"
- X509CertificateAdapter (as an implementation of the sub-interface AuthenticationMechanism)
- GSSUPAuthenticationMechanism (as an implementation of the sub-interface AuthenticationMechanism)

Methods

Virtual void initialize (const std::string& name, ::vbsec::InitOptions&) =0;
This method initializes the IdentityAdapter with the given name and 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.
vbsec::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 principle 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 method sets the privilege attribute for the identity.

Arguments
The privilege attribute to be set for the identity.

Exceptions none

virtual void deleteIdentity (::vbsec::Subject&) =0;
This method deletes the principals and the credentials associated with the specified target subject.

Arguments
The target subject for which the principals and the credentials recognized by this IdentityAdapter are to be deleted.

Exceptions none

vbsec::MechanismAdapter

Extending from IdentityAdapter, a MechanismAdapter has the additional capability of changing the target information. This is very useful in the case where the mechanism used in a remote site is not available locally. Therefore, the local identity must be adapted before sending to the remote site.

In the out-of-box installation of VisiSecure, there is no class direct implementation of MechanismAdapter, while a few classes implement the sub-interface AuthenticationMechanism, which in turn gives the support of this interface.
vbsec::AuthenticationMechanisms

This class represents a full-fledged mechanism which provides all the functionality needed to support an authentication mechanism in conjunction with the CSIv2 protocol.

Included with VisiSecure are the following implementations for GSSUP based and X509 Certificate based authentication mechanisms respectively:

- GSSUPAuthenticationMechanism
- X509CertificateAdapter

In addition to the methods inherited from its super interfaces, AuthenticationMechanism also has the following categories of methods defined.

Credential-related methods

Use these methods to acquire and/or destroy credentials.

```cpp
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

```cpp
virtual ::vbsec::Subject* acquireCredentials (const std::string& target, ::vbsec::CallbackHandler*) =0;
```

This method acquires credentials for a given string representation of the Target. The credentials acquired depend on the mechanism and the information it requires for authentication.

**Arguments**

This method takes the following two arguments:

- A string representation of the Target.
- The corresponding callback handlers used to communicate with user for acquiring the credential.

**Returns**

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

**Exceptions**

- none

```cpp
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

**createInitContext**

```cpp
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.

```cpp
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

```cpp
virtual ::CSI::GSSToken* createFinalContext (::vbsec::Subject&) =0;
```

This method creates a final context token to return to a client.

**Arguments**

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

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

\[
\text{vbroker.security.domain.<domain-name>.provider}
\]

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

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

### Methods

**virtual void initialize (const std::string& name, ::vbsec::InitOptions& options) =0;**

This method initializes an Authorization Services provider.

**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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORB</td>
<td>The ORB instance used for the current system.</td>
</tr>
<tr>
<td>Logger</td>
<td>A SimpleLogger instance used for login in the current system.</td>
</tr>
<tr>
<td>LogLevel</td>
<td>An integer value denoting the security logging level.</td>
</tr>
</tbody>
</table>

**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**

- 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.

Exceptions

none

vbsec::Privileges

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

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

The Privileges class implements the javax.security.auth.Destroyable interface.

Constructors

Privileges (const std::string& name, ::vbsec::Subject& subject);

This constructor creates a privileges object with the given name and associates it with the given subject.

Arguments

The method takes the following two arguments:

- Name of the Privileges object, which is actually the associated Subject's name.
- The target subject.

Exceptions

none

Methods

::vbsec::Subject& getSubject () const ;

This method returns the subject that the privileges object represents.

Returns

The target subject.

Exceptions

none

std::string getSubjectName () const ;

This method returns the name of the associated subject object.

Returns

The target subject.

Exceptions

none

const ::vbsec::ATTRIBUTE_MAP& getAttributes () const ;
vbsec::AttributeCodec

This method returns the attribute map of the user.

Returns
The user’s attribute map.

Exceptions
none

void setDBAttributes (const ::vbsec::ATTRIBUTE_MAP& map);

This method updates the DN Attributes of the user.

Arguments
The new DN Attributes Map.

Note
After the DN Attributes Map has been set, the Privileges object will set the underlying
DN Attributes Map as unmodifiable.

Exceptions
none

const ::vbsec::ATTRIBUTE_MAP* getDNAttributes() const;

This method returns the DN Attributes of the Privileges object, which can be null.

Returns
User’s DN Attributes map, which is not modifiable.

Exceptions
none

bool isDestroyed() const;

This method checks whether the privileges object has been destroyed or not.

Returns
true|false

Exceptions
none

std::string toString() const;

This method overrides the default toString implementation of java.lang.Object, and
returns “Privileges for <subject name>” information.

Returns
List of privileges for each subject name.

Exceptions
none

vbsec::AttributeCodec

The AttributeCodec objects are responsible for encoding and decoding privileges
attributes of a given subject. This allows clients and servers to communicate privilege
information to each other. Though the privilege information is used as the basis for the
Authorization decision-making process, AttributeCodec selection is based on the
information presented in the IOR published by the server. Inside the IOR, the server
publishes information on the encoding scheme supported, while clients select an
AttributeCodec that supports the given encoding.

All the AttributeCodecs implementations are registered with the IdentityServices, which
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 multiply implementations existing in one ORB, and each is addressed internally
using the name provided.

Arguments
This method takes the following arguments:

- A string of AttributeCodec implementation names.
- Provider options.
For the provider's options, the following additional information is also passed during the initialization:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORB</td>
<td>The ORB instance used for the current system.</td>
</tr>
<tr>
<td>Logger</td>
<td>A SimpleLogger instance used for the current system for the purpose of logging.</td>
</tr>
<tr>
<td>LogLevel</td>
<td>An integer value denoting the security logging level.</td>
</tr>
</tbody>
</table>

**Exceptions**

Throws InitializationException if initialization of this AttributeCodec object fails.

```cpp
virtual std::string getName() const =0;
```

This method returns the name of the provider implementation.

**Returns**
The provider name string.

**Exceptions**

none

```cpp
virtual CSIIOP::ServiceConfigurationList* getPrivilegeAuthorities() const =0;
```

This method returns a list of supported privilege authorities.

**Returns**
A list of privilege authorities.

**Exceptions**

none

4. ```cpp
virtual CSI::AuthorizationElementType getSupportedEncoding() const = 0;
```

This method returns the supported AuthorizationElementType type.

**Returns**
An AuthorizationElementType type.

**Exceptions**

none

```cpp
virtual bool supportsClientDelegation() const =0;
```

Returns whether this implementation supports ClientDelegation.

**Returns**
true | false

**Exceptions**

none

```cpp
virtual CSI::AuthorizationToken* encode (const CSIIOP::ServiceConfigurationList& privilege_authorities, vbsec::Privileges& caller_privileges, vbsec::Privileges& asserter_privileges) =0;
```

This method encodes privileges as AuthorizationElements. This method encodes the privilege attributes of the given caller and the given asserter, if there is one. It will extract the privilege information from the subject and privilege map of the caller and the asserter.

Additionally, an implementation of the AttributeCodec (if supports ClientDelegation) may choose to verify whether the asserter is allowed to assert the caller based on the client delegation information presented by this caller.

**Arguments**

This method takes the following arguments:

- A set of caller privileges attributes.
- A set of asserter privileges attributes.

**Returns**
Encoded caller and asserter privileges.

**Exceptions**

Throws NoDelegationPermissionException if the assertion is not allowed.

```cpp
virtual void decode (const ::CSI::AuthorizationToken& encoded_attributes, vbsec::Privileges& caller_privileges, vbsec::Privileges& asserter_privileges) =0;
```

This method decodes authorization elements and populates the corresponding privileges objects. This is the inversion process of the encode method. When a server receives a set of encoded AuthorizationElements, it passes these elements to the
vbsec::Permission

AttributeCodec for interpretation. Based on the encoding method, one particular AttributeCodec consumes the attributes it understands. It may update the caller's or asserter's Privileges, or may add RolePermission directly to the subject's public credentials.

Arguments
This method takes the following arguments:
- A set of encoded Authorization Elements.
- A set of caller privileges.
- A set of asserter privileges.

Returns
This method returns nothing. Upon a successful processing, this AttributeCode object updates the caller's or asserter's Privileges maps as appropriate based on the information available in the authorization elements.

Exceptions
Throws NoDelegationPermissionException if the assertion is not authorized.

vbsec::Permission

Permission represents the authorization information to access resources. Every permission has a name, which can be interpreted only by the actual implementation.

Include file
The vbsecspisshared.h file should be included when you use this class.

Methods

bool implies (const Permission& p) const
Evaluate if the permission implies another given permission. This is used during the authorization process to determine if the caller permissions imply the permissions required by the resource. Access will be granted if the caller permissions imply the required permission, or denied if not.

Arguments
the permission p to be evaluated.

Returns
true if the permission implies an existing permission, false otherwise.

bool operator==(const Permission& p) const
Checks if the permission equals another given permission.

Arguments
the permission p to be evaluated.

Returns
true if the permissions are equal, false otherwise.

std::string getName () const
Gets the name of the permission.

Returns
the name of the permission.

std::string getActions () const
Get the actions of the permission as a string. It is only interpreted by the actual implementation.

Returns
The string representation of the action for the permission.

std::string toString () const
Get the string representation of the permission.

Returns
The string representation of the permission.
vbsec::PermissionCollection

PermissionCollection represents a collection of permissions.

Include file

The `vbsecsppishared.h` file should be included when you use this class.

Methods

```cpp
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

```cpp
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

```cpp
virtual bool implies (const Permission& permission) const;
```

This method checks whether the permission object passed in implies this RolePermission object. The check is based on strict equality, as the method only returns `true` (implies) when ALL the following conditions exist:

1. the permission object given is an instance of RolePermission, and
2. the name of the permission object given equals the name of this RolePermission.

**Arguments**

- A Permission object to check.

**Returns**

- `true` | `false`

**Exceptions**

- none

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

This method returns the action associated with this RolePermission.

**Returns**

- Always returns null, since there are no actions associated with a RolePermission object.

**Exceptions**

- none
When a remote peer (server or process) makes identity assertions in order to act on behalf of the callers, the end-tier server needs to trust the peer to make such assertions. This is meant to prevent untrusted clients from making assertions.

The key method is `isAssertionTrusted`, which is called to determine whether the assertion is trusted given the caller subject and asserter's privileges. This method is called (by the underline implementation) after the corresponding authorization elements transmitted from a client to the server have been consumed.

You use the TrustProvider class to implement trust rules which determine whether the end-tier server accepts identity assertions from a given asserting subject. The TrustProvider class is very closely related to the implementation of the AttributeCodec objects and the privileges. For example, it is possible to provide the decision-making implementation as follows:

1. Provide class implementations representing a proxy endorsement attribute,
2. AttributeCodec implements the necessary logic then passes the attributes and imports them to the caller subject on the server-side. It is also necessary to return `true` for the method `supportsClientDelegation` defined in the AttributeCodec interface.
3. Provide the method implementation based on the proxy endorsement attribute of the caller and the privileges of the asserter.

This type of evaluation of trust, which is based on rules provided by the caller, is referred to as Forward Trust. Backward Trust is when the evaluation of trust is based on the rules of the target. Backward Trust is the default provided with the VisiSecure installation. For more information, see “Trust assertions and plug-ins” on page 21.

### Methods

```cpp
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORB</td>
<td>The ORB instance used for the current system.</td>
</tr>
<tr>
<td>Logger</td>
<td>A SimpleLogger instance used for the current system for the purpose of logging.</td>
</tr>
<tr>
<td>LogLevel</td>
<td>An integer value denoting the security logging level.</td>
</tr>
</tbody>
</table>

#### Exceptions

Throws `InitializationException` if initialization of the TrustProvider fails.

```cpp
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**

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

**Data Members**

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

A string map containing name/value pair presenting parsed property setting.

```cpp
::PortableInterceptor::ORBInitInfo* initInfo
```

Object representing the ORB initialization information.

```cpp
::IOP::Codec* codec
```

An IOP Codec object.

```cpp
::vbsec::SimpleLogger* logger
```

A logger object.

```cpp
int logLevel
```

The log level currently configured for the security service.

**vbsec::SimpleLogger**

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

**Include file**

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

**Methods**

```cpp
::std::ostream& WARNING()
```

Returns the logging output stream for warning messages.

**Returns**

The logging output stream for `LEVEL_WARNING`

```cpp
::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`
vbsec::SimpleLogger

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
Index
### C

- C++ applications
  - providing security identities 71
  - securing 71
  - setting QoP 73
- C++, security APIs 95
- CA 14, 15
  - distinguished name 15
  - revoked certificates 16
  - certificate authorities 15
  - Certificate Authority 14
  - Certificate Authority (CA)
    - Certification Revocation List (CRL) 16
    - revoked certificate serial numbers 16
- certificate file, Apache web server 78
- certificate mechanism 38
- Certificate Passthrough
  - enabling 81
    - IIOP configuration directive 82
    - mod_ssl directive 78
  - to Borland web container 81
- certificate request 14
- certificate, cipher suites 17
- certificates 14
  - chains 15
  - creating 14
  - distinguished name 15
  - generated files 14
  - generating 14
  - obtaining 14
  - public key 16
  - using at the SSL layer 38
- Certification Revocation List
  - creating 16
  - file format 16
  - VisiSecure for C++ 16
- cipher suites 17
  - supported 17
- CipherSuiteName 104
- cipher-text 13
- clear-text 13
- CN 45
- commands, conventions 4
- config.jaas 30
  - use with security profiles 54
- constructors
  - Privileges class 127
  - RolePermission class 131
  - vbsec::Privileges 127
  - vbsec::RolePermission 131
- CORBA authorization 47
  - setting up 48
- CORBAsec
  - X509Cert 110
  - X509CertExtension 112
  - credentials 26
  - CRL 16
csv2
  - AccessPolicyManager 115
  - ObjectAccessPolicy 115

### D

- delegation 20
- Developer Support, contacting 4
- digital signatures 14
- distinguished name 15
- DN 15
- DNAAdapter 121
- documentation 2
  - .pdf format 3
  - accessing Help Topics 3
  - Borland Security Guide 2
  - on the web 5
  - platform conventions used in 4
  - type conventions used in 4
  - updates on the web 3
  - VisiBroker for .NET Developer’s Guide 2
  - VisiBroker for C++ API Reference 2
  - VisiBroker for C++ Developer's Guide 2
  - VisiBroker for Java Developer's Guide 2
  - VisiBroker GateKeeper Guide 3
  - VisiBroker Installation Guide 2
  - VisiBroker VisiNotify Guide 2
  - VisiBroker VisiTelcoLog Guide 2
  - VisiBroker VisiTime Guide 2
  - VisiBroker VisiTransact Guide 2
- domain name
  - 87
  - domain, security profiles for 63
  - domain_name
  - 87, 91
- domains
  - authorization 46
  - Run-as 47
  - VisiBroker domain authorization properties 47
- DS 14

### E

- encrypted hash 14
- encryption
  - asymmetric encryption 13
  - public-key 13
  - setting level of 17
  - symmetric encryption 14
- exceptions, Security Provider Interface for C++ (SPI) 119

### F

- files, certificate 14
- formatted target 37
- forward trust 21

### G

- GROUP 45
- GSSUP mechanism 37
- GSSUPAuthenticationMechanism 121, 123

### H

- hashes, encrypted 14
- Help Topics, accessing 3
- Host LoginModule 32  
  - code sample 36
  - realm entry syntax 35
- HTTPS 22
identities
setting up 36
setting up assertion 68, 73
ways to provide 66, 71
identity assertion
backward trust 21
delegation 20
forward trust 21
impersonation 20
trusting assertions 21
identity assertions 19
plug-ins 21
TrustProvider interface 21
IdentityAdapter 119, 121
AnonymousAdapter 121
DNAAdapter 121
GSSUPAuthenticationMechanism 121
X509CertificateAdapter 121
IIOP connector, enabling Certificate Passthrough 82
IIOP over HTTPS 22
Microsoft IE 22
Netscape Communicator 22
impersonation 20
ISO X.509 14, 15

JAAS 11, 25
JSSE and 65
pluggable authentication 27
security profiles and 54
JAAS authentication 25
credentials 25
principals 25
subjects 25
JAAS authentication credentials
private 26
public 26
Java applications
providing security identities 66
securing 65
setting QoP 67
Java Authentication and Authorization Service (JAAS) 11
JDBC LoginModule 32
code sample 34
properties 34
realm entry syntax 34
JSSE 17
basic concepts 65
JAAS and 65
JSSE X509TrustManager 16

key file, Apache web server 78

LDAP LoginModule 32
properties 35
Realm Entry syntax 35
logical operators for rules 45
LoginContext class 27
LoginModule
and realm 30
config.jaas 30
LoginModule interface 27
LoginModules 27
authentication 28
authentication mechanisms 27
BasicLoginModule 32
Borland provided 32
commit phase 29
Host LoginModule 32
JDBC LoginModule 32
LDAP LoginModule 32
realm 29
stacked 28

MechanismAdapter interface 122
method_name gt 87
methods
AttributeCodec interface 128
AuthenticationMechanisms interface 123, 124
AuthorizationServicesProvider interface 126
IdentityAdapter interface 121
MechanismAdapter interface 123
Privileges class 127
Resource interface 127
RolePermission class 131
Target interface 125
TrustProvider interface 132
vbsec::AttributeCodec 128
vbsec::AuthenticationMechanisms 123, 124
vbsec::AuthorizationServicesProvider 126
vbsec::IdentityAdapter 121
vbsec::MechanismAdapter 123
vbsec::Privileges 127
vbsec::Resource 127
vbsec::RolePermission 131
vbsec::Target 125
vbsec::TrustProvider 132
mod_ssl
Apache web server security 77
directive 80
directives 77

Newsgroups 5

n gt 87, 91
mod_ssl
Apache web server security 77
directive 80
directives 77

Index 137
security (C++)
- AttributeCodec 119, 128
- AttributeCodec methods 128
- AuthenticationMechanism 119
- AuthenticationMechanisms 123
- AuthenticationMechanisms methods 123, 124
- AuthorizationServicesProvider 126
- AuthorizationServicesProvider methods 126
- com.borland.security.spi.IdentityAdapter 121
- IdentityAdapter 119, 121
- IdentityAdapter methods 121
- MechanismAdapter 122
- MechanismAdapter methods 123
- Privileges class 127
- Privileges constructors 127
- Privileges methods 127
- Resource 127
- Resource methods 127
- RolePermission 131
- RolePermission constructors 131
- RolePermission methods 131
- Service Provider Interface (SPI) 117
- Service Provider Interface (SPI) exceptions 119
- SPI exceptions 119
- SPI provider settings 119
- Target 125
- Target methods 125
- TrustProvider 119, 132
- TrustProvider methods 132
- vbsec::AttributeCodec 119
- vbsec::AuthenticationMechanisms 123
- vbsec::MechanismAdapter 122
- vbsec::RolePermission 131
- vbsec::Target 125
- vbsec::TrustProvider 132

security (Java)
- AuthenticationMechanism 12
- distributed environment 11, 12
- Security Provider Interface (SPI) 11
- SPI 11, 12
- security profiles 51
  - associating with domains 63
  - authentication 54
  - authorization 56
  - config.jaas and 54
  - creating 51
  - properties 62
  - rolemaps 56
  - VisiSecure properties 62
- security properties (C++) 91
- security properties (Java) 87
- Security Provider Interface for C++ (SPI)
  - AttributeCodec 119, 128
  - AuthenticationMechanism 119
  - AuthenticationMechanisms 123
  - AuthorizationServicesProvider 126
  - IdentityAdapter 119, 121
  - MechanismAdapter 122
  - Privileges class 127
  - Resource 127
  - RolePermission 131
  - Target 125
  - TrustProvider 119, 132

Security Provider Interface for Java (SPI) 11
- TrustProvider interface 21
- serial numbers, revoked 16
- server identification 36
- server, identity assertions 19
- ServerQoPPolicyImpl 113
- Service Provider Interface for C++ (SPI) 117
- signatures, digital 14
- Software updates 5
- SPI 11
- AttributeCodec 119
- SPI (C++) 117
- AuthenticationMechanism 119
- AuthenticationMechanisms 123
- AuthorizationServicesProvider 126
- exceptions 119
- IdentityAdapter 119, 121
- MechanismAdapter 122
- modules 119
- Privileges class 127
- provider settings 119
- Resource 127
- RolePermission 131
- Target interface 125
- TrustProvider 119, 132
- SPI (Java) 12
- SSL
  - authentication, Borland web container 81
  - cipher suite 17
  - communication 17
  - encryption 17
  - examining information 68, 74
  - layer, using certificates 38
  - ssl
    - CipherSuiteInfo 104
    - Current 106
- subjects 25
- Support, contacting 4
- symbols
  - brackets [ ] 4
  - ellipsis ... 4
  - vertical bar | 4
- symmetric encryption 14

T
- Target interface 125
- Technical Support, contacting 4
- temporary privileges 21
- three-tier authorization 83
- Tomcat web container
  - enabling Certificate Passthrough 81
  - https type connections 81
- security 83
- three-tier authorization 83
- trust
  - backward 21
  - forward 21
  - identity assertion 68, 73
  - setting 68, 73
- Trust Management 16
- TrustProvider 119
- TrustProvider interface 132
U
user domain 9
usernames, authentication 12

V
vault 38
creating 38
VaultGen tool 38
VaultGen example 40
VaultGen tool 38
vbroker.se.iop_tp.scm.ssl.listener.trustInClient
property 87
vbroker.security
alwaysSecure property 87, 91
assertions.trust.&lt 87, 91
assertions.trust.all property 87, 91
authDomains property 87, 91
authentication.callbackHandler property 87, 91
authentication.clearCredentialsOnFailure
property 87
authentication.config property 87, 91
authentication.retryCount property 87, 91
cipherList property 87, 91
controlAdminAccess property 87
CRLRepository 16, 91
defaultJSSETrust property 87
disable property 87, 91
domain.&lt 87, 91
enableAuthentication property 87
identity.enableReactiveLogin property 87
identity.reactiveLogin property 91
identity.reauthenticateOnFailure property 87
logFile property 91
login property 87, 91
login.realms property 87, 91
logLevel property 87, 91
peerAuthenticationMode property 87, 91
requireAuthentication property 87, 91
secureTransport property 87, 91
server.requireUPIdentity property 87, 91
server.transport property 87, 91
serverManager.authDomain property 87
serverManager.role.&lt 87
serverManager.role.all property 87
support.gatekeeper.replyForSAS property 87
transport.protocol property 87
trustpointsRepository property 87, 91
vault property 87, 91
wallet.identity property 87, 91
wallet.password property 87, 91
wallet.type property 87, 91
vbroker.security.authentication.config property 36
vbroker.security.callbackhandler property 36
vbroker.security.login property 36
vbsec::CertificateFactory 109
vbsec::ClientConfigImpl 114
vbsec::ClientQoPPolicyImpl 114
vbsec::Context 96
vbsec::Credential 99
vbsec::Current 95
vbsec::IdentityAdapter 121
methods 121
vbsec::MechanismAdapter 122
methods 123
vbsec::Principal 99
vbsec::Privileges 127
constructors 127
methods 127
vbsec::Resource 127
methods 127
vbsec::RolePermission 131
constructors 131
methods 131
vbsec::SecureSocketProvider 105
vbsec::ServerConfigImpl 113
vbsec::SSLSession 103
vbsec::Subject 100
vbsec::Target 125
methods 125
vbsec::TrustProvider 132
methods 132
vbsec::VBSSLContext 104
vbsec::Wallet 101
vbsec::WalletFactory 101
VisiBroker for C++, security properties 91
VisiBroker for Java, security properties 87
VisiBroker overview 1
VisiSecure 8
C++ APIs 95
properties and security profiles 62
VisiSecure API (C++)
AccessPolicyManager 115
authorization API 114
Certificate APIs 109
CertificateFactory 109
CipherSuiteInfo 104
CipherSuiteName 104
class Credential 99
ClientConfigImpl 114
ClientQoPPolicyImpl 114
Context 96
Current 95, 106
general APIs 95
ObjectAccessPolicy 115
Principal 99
QoP APIs 112
SecureSocketProvider 105
ServerConfigImpl 113
ServerQoPPolicyImpl 113
SSL APIs 102
SSLSession 103
Subject 99
VBSSLContext 104
Wallet 101
WalletFactory 101
X509Cert 110
X509Cert Extension 112
VisiSecure APIs (C++) 95
VisiSecure for C++, Certification Revocation List 16
VisiSecure SPI (C++) 117
  AttributeCodec 119, 128
  AttributeCodec methods 128
  AuthenticationMechanism 119
  AuthenticationMechanisms 123
  AuthenticationMechanisms methods 123
  AuthorizationServiceProvider 126
  exceptions 119
  IdentityAdapter 119, 121
  IdentityAdapter methods 121
  MechanismAdapter 122
  MechanismAdapter methods 123
  Privileges 127
  Privileges constructors 127
  Privileges methods 127
  Providers 119
  Resource 127
  Resource methods 127
  RolePermission 131
  RolePermission constructors 131
  RolePermission methods 131
  SPI modules 119
  Target 125
  Target methods 125
  TrustProvider 119, 132
  TrustProvider methods 132

VisiSecure SPI(C++), Service Provider Interface (SPI)
  provider settings 119

W
  web application security 83
  web components, security 77, 83, 84
  web container, security 83
  wildcard assertion, code sample 46
  wildcard assertions 45
  World Wide Web
    Borland documentation on the 5
    Borland newsgroups 5
    Borland updated software 5

X
  X.509 certificates 15
  X509CertificateAdapter 121, 123