Voyager Security Developer's Guide

Version 1.2 for Voyager 8.0

Table of Contents

Introduction

Prerequisites

Voyager™ Security Developer's Guide assumes a basic working knowledge of Java/CSharp and Voyager, as well as experience developing network applications. Voyager Security requires Voyager Version 6.0 or later.

Overview

A security policy knows the entities that are recognized within a security domain, and rights or privileges that are granted to that entity within that domain. An entity authenticates against a policy, and application code verifies that a permission or privilege was granted to the entity (principal) associated with the current thread.

Voyager Security provides a framework and implementation for creating a secure distributed application through a policy-based mechanism.

Authentication and permission verification is done through a standard interface provided by the Voyager Security Application Program Interface (API). The API uses a security policy provider for the actual implementations and mechanisms for principal, authentication and permission verification. The Voyager platform can use any form of security and its related primitives. These mechanisms can range from simple user and password authorization to public/private key certificates.

Contacting Technical Support

Recursion Software welcomes your problem reports, and appreciates all comments and suggestions for improving Voyager. Please send all feedback to the Recursion Software Technical Support department.

Technical support for Voyager is available via the email and phone. You can also contact Technical Support by sending email to psupport@recursionsw.com or by calling (972) 731-8800.

Language Security

Security Basics

The Voyager Security Basics chapter details the following:

- Common terms
- Establishing trust in code and executor
- General security concepts

Common Terms

The following definitions define common terms and concepts used in the Voyager Security Developer's Guide.

Security Domain - a security domain is an enterprise-wide set of computer systems that share a common security policy and use the same security mechanisms. For example, a Microsoft Domain is a security domain.

Entity - an entity can be a person, machine or organization. All code executes on behalf of some entity, either implicitly or explicitly. When a user starts a web browser, the web browser acts on behalf of that user.

Principal - a principal is an application object or structure that represents an entity. A principal identifies the entity it represents in some form, such as a public key certificate or a username.

Identity - an identity is information that describes an entity. An identity is represented by a principal. Generally, a public key certificate has the information necessary to establish the identity of an entity.

Security Context - a context is information or an environment that gives meaning to something else. In the case of security, a security context is the association between a thread of execution and the entity (principal) or entities for which the thread is executing.

Signer - a signer implies a particular type of principal, one used to generate digital signatures. In the case of Java, Java classes can be signed by an entity vouching for the safety of the classes. The entity that signed these classes is represented as a signer in the application.

Credential - a credential is information that establishes the identity of an entity, generally during authentication or login.

Establishing Trust in Code and Executor

Two main concerns of trust exist with regard to services and applications.

- Does the developer trust the code being executed?

The Java security sandbox model, typically used with Applets, was designed to address the code trust concern. Classes are trusted when they are signed by a trusted entity.

- Does the developer trust the user on whose behalf the code is acting?

The method that addresses this concern, not provided by the Java Development Kit (JDK), allows association of a thread of execution with a particular entity. Thread association can be accomplished by creating a context that travels with the thread through the stack and across virtual machines.

Java Security Sandbox Model

The Java security sandbox provides boundaries within which untrusted code must execute. These boundaries are enforced by installing a `java.net.SecurityManager` subclass used by the Java Virtual Machine (JVM). Only one `SecurityManager` is installed per JVM.

Trust is usually determined by scanning the classes on the current thread execution stack to verify which classes are deemed as *trusted*. Currently, the only mechanism available to group classes into degrees of trust is to tag a class with a particular signer object. A signer object, or signer, represents an entity that vouches for a given resource or class. If the application trusts a particular entity, for example, `Recursion Software`, and Recursion Software signed a particular set of classes that will be used in the application, then the application should *trust* those classes. Signers are only assigned within the `ClassLoader` that defined the class bytes. Classes loaded by the primordial classloader (null classloader) have implicit trust and are not assigned a signer.

For example, when a piece of code is executed and the code attempts to perform some action governed by the installed `SecurityManager`, it is the responsibility of the `SecurityManager` to determine whether any of the code on the stack was loaded by a `ClassLoader`, and if so, who signed the code and whether the entity associated with the signer object is trusted. If the invocation is deemed untrusted, a `SecurityException` is thrown to abort the operation.

Execution Context Model

By allowing an entity to be associated with a thread of execution as it traverses local and remote JVMs, application code can, at any point, verify whether that entity has the privileges necessary to perform a particular operation. Trust in an entity is initially established when it authenticates itself with the security domain by providing domain-specific credentials, or information that proves its identity to the domain, such as a password.

In the context model, an authenticated entity is represented by a principal. The principal in turn is associated with the thread of execution via a context. As the thread executes methods on remote objects, the associated context moves transparently with the invocation across virtual machines. A principal is typically a concrete implementation of the `java.security.Principal` interface, unlike the generic signer object of type `java.net.Object` used by the sandbox model.

Additionally, a context can be associated with a proxy or JVM instead of a particular thread. All method invocations from a proxy or JVM pass the context associated with it to the remote virtual machine. When the context arrives at the remote host (JVM), it is associated with the current thread of execution.

Understanding Security Concepts

The following sections detail important security concepts, such as enforcing policy, data privacy and integrity, non-repudiation of resource utilization, secure sections and privilege delegation.

Data Privacy and Integrity

The Java language and virtual machine protect in-memory data from direct memory access and offer byte-code verification during class loading. Refer to the Java Virtual Machine Specification and the Java Language Specification for more information. These specifications define both the Java language and the virtual machine that interprets the language.

The .NET Framework provides code access permission classes, each of which encapsulates the ability to access a particular resource. You use these permissions to indicate to the .NET Framework what your code needs to be allowed to do and to indicate what your code's callers must be authorized to do. Policy also uses these objects to determine what permissions to grant to code.

To protect data as it moves over a public network, Voyager allows implementation of the Secure Sockets Layer (SSL) protocol to provide both data privacy and data integrity during transmission. Data-integrity checks guarantee that any information sent between hosts has not been tampered with. Data privacy (or encryption) prevents data from being read at any point between two hosts. SSL also offers the ability to establish trust between hosts by exchanging client and server public-key certificates. A simple, custom SSL transport implementation can provide all of these features.

Non-Repudiation of Resource Utilization

Non-repudiation is the process by which a non-refutable record of user activity is maintained. The process identifies the user accessing resources and services provided by the server (host). Generally, simple logging mechanisms are sufficient for this purpose.

Secure Sections

Secure sections allow temporary replacement of the current security execution context, so that the enclosed code executes on behalf of the new principal. The new context remains in scope within the section and propagates between virtual machines, if necessary. For example, if a remote client with limited rights invokes a simple database query service, it may not have the required privileges to perform the query. If the query was executed on behalf of a second (intermediate) user with sufficient privileges, the client could then retrieve the necessary result set without having been explicitly granted the necessary permissions. These new rights or privileges only remain in effect inside the scope of the secure section.

A secured section acts as a delegation boundary. When a secured section is entered, the policy can delegate rights from the current principal to the new principal for which the enclosed section is executing.

Privilege Delegation

Privilege delegation occurs when a calling entity authorizes an intermediate entity to perform a task using a set of the rights granted to the calling entity. For delegation, the underlying security mechanisms must be notified that a boundary has been crossed, that the calling entity (principal) is no longer in scope. This occurs only when entering a secure section.

Privilege delegation modes include the following:

- **None** - when a context reaches a delegation boundary, it is replaced with the intermediate's security context until it returns into scope.
- **Simple** - the current security context is left intact.
- **Cascaded** - the current security context is merged with the intermediate entities' privileges, and a new context is propagated along the thread of execution.

The current policy is responsible for enforcing one of these delegation modes.

For example, an Enterprise Java Bean (EJB) container is responsible for providing the boundary between a client and a bean by performing all method invocations within a secure section. The EJB specification requires two delegation modes be supported: `none` and `simple`. These are the `run as` modes specified during bean deployment.

Security Library

Overview

The Voyager security library includes multiple components. The following chapter describes the Voyager security framework, library and how to implement a custom policy that presents an interface to an enterprise-wide security domain for use by application code running on top of Voyager. The Security Library chapter details the following:

- Integrating a policy facade
- Understanding the Voyager Security architecture
- Using the security interface
- Working with security policies
- Using signed classes
- Working with VoyagerSecurityManager

Integrating a Policy Facade

Figure 1 shows the integration of a policy facade into each host virtual machine (VM). In this case, the actual security domain, a remotely available access control list (ACL), actually resides in the enterprise system. The policy implementation is responsible for keeping groups, users and permissions cached locally for performance.

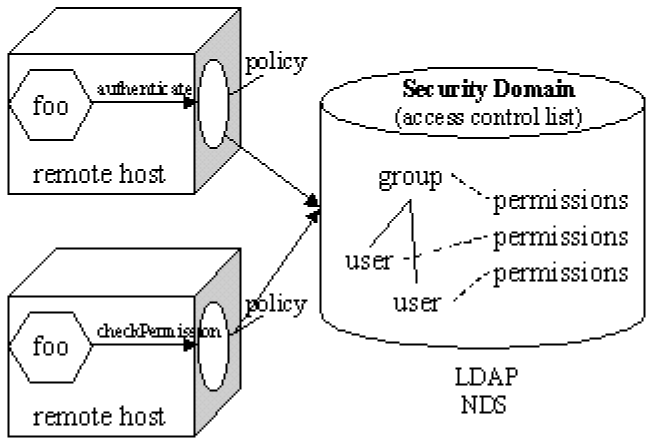

Figure 1 - Policy facade

Figure 2 shows the integration of all the components of the Voyager security library with the Voyager and any domain-specific applications.

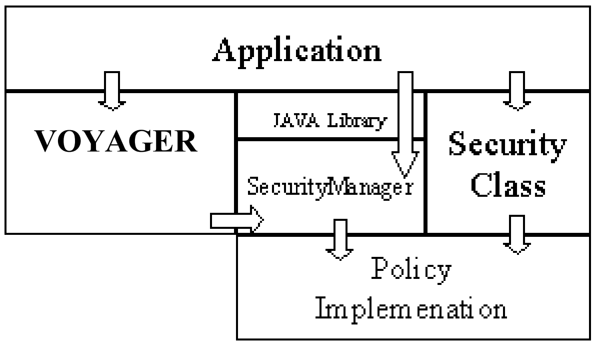

Figure 2 – Voyager Security Library/Voyager integration

A given application may implicitly invoke `Policy` permission checks via an installed `SecurityManager`, or it may explicitly verify permissions against the `Security` class Application Program Interface (API).

Understanding the Voyager Security Architecture

All Voyager security calls and functionality begin through the Voyager `Security` class API. At startup, the `Security` class initializes the system by loading any specified policies and setting the security owner, allowing many additional properties to be set or modified after startup.

Every company or enterprise-wide security domain maintains a policy that recognizes specific entities and manages the privileges granted them. By subclassing the Voyager security library's `Policy` abstract class and implementing the `SecurityContext` interface, the security domain can be represented in the application. Given a particular `Policy` implementation, application code can authenticate entities (principals) and verify privileges (permissions). `Policy` implementations are managed by the Voyager library `Security` class.

Optionally, the Voyager security library's `PolicyManager` abstract class can be implemented. `PolicyManager` allows for management of the underlying policy. For example, new users can be created and added, and new permissions may be associated with them. By default, the method `Policy.getPolicyManager()` returns null.

The `Security` class is also responsible for managing principal associations to threads and proxies, protecting them from application code and propagating them reliably between Java virtual machines (JVMs). To facilitate this goal, authenticated `Principal` instances are associated with `SecurityContext` interface implementations. The association is maintained by the `Policy` implementation, and in many cases, the `SecurityContext` implementations encapsulate (wrap) their associated `Principal`. The Voyager library `Security` class manages the associations between `SecurityContext` implementations and related threads and proxies, and it is this `SecurityContext` implementation that is propagated between JVMs. The `Principal` instance is only propagated when the `SecurityContext` implementation was designed to do so.

The `Security` class has many methods that are duplicated on the `Policy` class, like `authenticate()` and `checkPermission()`. All of the methods on the `Security` class delegate to the default policy and are offered as a convenience to application developers. If application code needs to interact with a policy other than the default, alternate policies can be retrieved. Policies are described in detail in Working with Security Policies.

Using the Security Interface

The following section describes how to use the security interface to perform the following procedures.

- Installing policy implementations
- Authenticating entities
- Verifying permissions
- Determining security owner
- Establishing client identity

Installing Policy Implementations

During the initialization of the `Security` class, a security policy is instantiated from the class name associated with the system property key `recursionsw.security.policy.1`, and set as the default. Set the property key via a properties file or programmatically before `Voyager.startup()` is invoked. For example, install a default policy by including the following line in a startup properties file.

```
recursionsw.security.policy.1=com.acme.marketing.SimplePolicy
```

Policies associated with `keys.2` or above are considered alternate policies and can be retrieved by name through the `Security.getPolicy( String name )` method.

Authentication

The `Security.authenticate( Object identity, Object credentials )` or `( Policy.authenticate( Object identity, Object credentials ) )` method is used to authenticate an entity's identity. Depending on the implementation, credentials are supplied that are verified by the policy provider. The `authenticate()` method returns an implementation of the `java.security.Principal` interface used to identify the user. The `Principal` may eventually be encapsulated by a `SecurityContext` implementation and passed between virtual machines for re-authentication, rights delegation and permission verification. For example:

```
try
  {
  Principal me = Security.authenticate( "chris",
"w1ldcat" );
  }
catch( Authentication exception )
  {
  System.out.println( "i cannot login" );
  }
```

The `Security.unauthenticate( Principal principal )` method allows a `Principal` to be un-authenticated. Some systems may provide tokens that represent the authenticated identity. In many cases, the system must destroy the tokens when they are no longer in use.

Permission Verification

The `Security.checkPermission( Object permission )` or `(Policy.checkPermission( Object permission) )` method verifies whether the current `Principal` has the requested permission. The `Principal` that permissions are checked against is retrieved from the security context associated with the current thread. If the underlying policy determines the given `Principal` does not have the necessary permissions, the `Policy` subclass must throw a `java.net.SecurityException` (or subclass). A permission can be of type String, an implementation of `java.security.acl.Permission` or any other type specific to the domain. For example, to verify permissions against the default policy, execute the following:

```
public Hashtable getHashtable() { Security.checkPermission(
new InvocationPermission( "Foo.getHashtable" ) ); return
hashTable; }
```

The method calling `getHashtable()` must catch a `SecurityException` if thrown, because the calling entity was not granted the `Foo.getHashtable InvocationPermission`.

Security Owner

Some methods on the `Security` class require specifying an owner. The security owner is set when the instance method `Security.setSecurityOwner( Principal principal )` is invoked. `Security.setSecurityOwner( Principal principal )` is only set by the method `Policy.initSecurity( Security security )` on the default policy during startup and cannot be changed. Depending on the domain, the owner can represent a single user or a group. `Policy.areEquivalent()` is called to test whether the security owner is the same as the one specified as an argument.

If a security owner is not set during initialization, all owner checks are ignored and allowed to pass. If an owner is specified as an argument, it must be compatible with the default `Policy`, the same provider that installed the security owner during initialization.

For example, the method `Security.defaultInvokeAs( Principal owner, Principal principal )` sets the default user. All method invocations propagate to remote JVMs when not specified by other means. `Security.defaultInvokeAs( Principal owner, Principal principal )` can easily cause a system to fail if any application code was allowed to set a random default principal. Any code that tries to set or modify this value must also provide a reference to the security owner set during startup, proving it is acting on behalf of an entity allowed to make such sensitive calls.

Establishing Client Identity

Before a secured method invoke call on a proxy, the local (client) JVM must establish the identity it is representing and associate it with the impending method invocation for use by the remote (server) JVM. The target method on the server retrieves the associated identity and verifies that the calling entity has the required permission.

There are a number of ways for the client to associate its identity (principal) with the method being invoked to inform the server (remote JVM) of the entity and method being represented. The client must first prove it is representing a valid and recognized entity. To do this it calls `Security.authenticate()` to retrieve an authenticated principal.

Associate the principal with the current thread by creating a secured section. Also, a single principal can be assigned to the local JVM, a proxy, or a `ThreadGroup` to act as the default for all method invocations from the local JVM. On the server side, the principal for a client is determined from the `SecurityContext` that was passed across the network connection from the client. Alternately, the connection may be capable of providing the identity of the remote JVM, or, if no context is available, a default principal for all new connections can be set. These capabilities are shown **Figure 3** and detailed in the sections that follow.

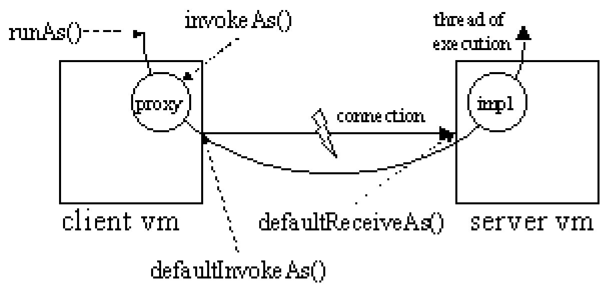

Figure 3 - Client Identity Associations

Secured Sections and Privilege Delegation

By invoking the `Security.runAs( Principal principal, SecureAction action )` or `Security.runAs( Principal principal, SecureExceptionAction action )` method, a secure section is created associating the given `Principal` with the current thread of execution. Code that must execute as the given `Principal` must be wrapped in an implementation of the `SecureAction` or `SecureExceptionAction` interfaces.

The `SecureAction` and `SecureExceptionAction` interfaces only declare one method, `public Object run()` for the `SecureAction` interface and `public Object run()` throws `Exception` for the `SecureExceptionAction` interface. Any return value is passed back to the calling method as type `java.net.Object`. For example:

```
Principal me = Security.authenticate( "chris", "w1ldcat" );
RecordSet set = (RecordSet) Security.runAs( me, new
SecureAction()
  {
  public Object run()
    {
    return dataProxy.getRecordSet();
    }
  });
```

The code example first authenticates the user `chris`, then invokes the method `data Proxy.getRecordSet()` on behalf of the authenticated user.

Before invoking a `SecureAction` or `SecureExceptionAction` implementation, the Voyager Security service determines whether the old principal associated with the current thread and the new principal are from the same policy. If so, the `Policy` implementation can opt to delegate any rights from the old principal to the new principal. The new principal (its `SecurityContext`) is then associated with the current thread until the `SecureAction.run()` or `SecureExceptionAction.run()` method returns.

Setting Default Client Identity on the Client Virtual Machine

The client's identity is set by calling one or more of the following methods.

- `Security.defaultInvokeAs( Principal owner, Principal principal )` - the client user is identified by a default context that propagates on every method invocation from the client virtual machine. The security owner must be specified in order to set the VM-wide user. The given `Principal` is not associated with a thread until the message from the local JVM reaches the remote JVM.
- `Security.invokeAs( Proxy proxy, Principal principal )` or `Policy.invokeAs( Proxy proxy, Principal principal )` - the given proxy can be associated with a security context so that all future method invocations from the client virtual machine represent the authenticated entity. The given `Principal` will not be associated with a thread until the message from the proxy reaches the remote JVM.
- `Security.runAs( Principal principal )` - a `SecurityContext` associated with the default policy and given principal is associated with the current thread. All method invocations in the local JVM and in any remote JVM act on behalf of the given `Principal`. By calling this method a second time with null as the principal value, the current context is removed. No delegation of rights occurs and, any `SecurityContext` previously associated with the current thread will be lost.
- `Security.runAs( Principal principal, ThreadGroup threadGroup )` - all method invocations from any thread in the given `ThreadGroup` has the given `Principal` associated with it when no other context is already associated with the thread or proxy invoking the message. `ThreadGroup.checkAccess()` is called to verify that the calling code or principal has the privileges required to make changes to the given `ThreadGroup`.

Setting Default Client Identity on the Server Virtual Machine

To establish a default client identity on the server side, perform one of the following:

If an unknown client is requesting access to a service or a security context, but is not propagated from a client, a default user can be specified on the server side for method invocations. This is analogous to a guest or 'unauthenticated user' identity. The method `Security.defaultReceiveAs( Principal owner, Principal principal )` sets the default guest user to the given `Principal`.

Working with Security Policies

The `Policy` implementation provides domain-specific implementations of `Principal` and permission, as well as performs all authentication and permission verification operations as applicable to the domain. The permission is not required to be `java.security.acl.Permission`. The policy also acts as a `SecurityContext` factory by returning instances of the `SecurityContext` interface that are associated with a given `Principal`.

Using Multiple Policies-Domains/Realms

Any JVM can support multiple policies to allow a particular host to act as a bridge to multiple security domains. For example, a simple client may use resources provided by a middle tier. This middle tier may in turn communicate to back-end resources, but for it to do so, it may need to act on behalf of a second privileged user that is a member of a domain different from the domain of the original user.

When alternate policies are installed, they are keyed to the value returned by `Policy.getPolicyName()` . To retrieve a particular policy, call `Security.getPolicy( String name )`, where the name is the same value as that returned by `Policy.getPolicyName()`. Additionally, the current policy associated with the `SecurityContext` implementation associated with the current thread can be retrieved by calling `Security.getCurrentPolicy()`.

Implementing a Custom Security Policy

To implement a custom policy, the `Policy` abstract class must be subclassed and the interface `SecurityContext` must be implemented. Optionally, the `PolicyManager` abstract class can be implemented. A number of methods declared on the `Policy` must be defined by the subclass. Some methods may have default implementations and may be optionally overridden depending on the domain.

Security Initialization

The method `Policy.initSecurity()` allows the default policy to safely set the security owner on the `Security` class. The security owner is the entity that governs access to certain security APIs. If a security owner is not set, any methods requiring a reference to the owner as an argument can be set to null.

After setting the security owner, set the system principal. The system principal is associated with system-level threads that need special rights to perform their tasks. For example, the distributed garbage collection thread needs to be associated with a principal that has rights on remote systems to perform necessary garbage collection procedures.

Security Context

A security context logically maintains an association between a thread of execution and a principal. The method of implementation is determined by the policy implementer. For example, a security context implementation may encapsulate the domain's notion of a principal, being responsible for protecting this principal from unauthorized access so that untrusted code cannot extract the principal and impersonate the entity it represents.

The security context is serialized to pass the association between virtual machines over the network. By creating `read/writeObject()` or `read/writeExternal()` methods, developers can control how the principal and any secrets it contains are transmitted over a public network.

The context also receives notification events when it has arrived on a remote virtual machine, via the `SecurityContext.arrived()` method. When notified, the associated principal should be re-authenticated on the new host (JVM). The policy is responsible for determining whether or not a principal entering the virtual machine has been authenticated for use inside the virtual machine.

The `Policy.getContextFor( Principal principal )` method returns a `SecurityContext` instance for the given `Principal`. The `Policy.getPrincipalFor( SecurityContext context )` method must return the same `Principal` when given the context returned by `getContextFor()`.

Principal Equivalence

Equivalency is a function of the security domain and is delegated to the `Policy`. Equivalency may mean that two principals represent the same entity, or that they may be considered equivalent when one principal represents a group and the second is a member of that group. Call `Policy.areEquivalent( Principal lhs, Principal rhs )`to test for principal equivalency.

PolicyManager

The `PolicyManager`, if implemented, allows user or domain code to manage the principals and permissions used by the policy. To get an instance of the `PolicyManager` class, the method `Policy.getPolicyManager( Principal administrator )` must be called, where the administrator is a principal with required privileges to add and remove users, and to grant and deny permissions. By default, this method on the policy returns null.

Using Signed Classes

In general, class bytes are trusted when they were retrieved from the local classpath, or when loaded outside of the classpath (from a remote server, for example) and signed by a trusted entity. A signer is typically an entity that vouches for the class bytes being provided.

Untrusted bytes are those generally loaded outside the classpath and without a known or trusted signer. Untrusted bytes may or may not be allowed to make certain calls restricted by the current `SecurityManager` subclass.

Associating Principals with Classes

The `VoyagerClassLoader` provides the functionality to associate principals with classes to establish their level of trust by providing custom implementations of the `IResourceLoader` interface. The `VoyagerClassLoader` determines whether or not signers are available from the resource loader in which it found class bytes and associates them with the class. An installed `VoyagerSecurityManager` subclass can decide whether to allow a secured method invocation. The `java.net.SecurityManager` has a number of system-call checks that can be secured from invocation by untrusted code.

The interface method `IResourceLoader.getSigners()` returns an array of signer objects that vouch for the classes the resource loader supplies. Only one set of signers should be associated with a resource loader implementation because a resource loader should retrieve bytes from one source, like a remote HTTP server or jar file. In some cases, the array of signers would be a certificate chain that can be validated with a local trusted copy of the domain root certificate.

Understanding VoyagerSecurityManager

Generally the `VoyagerSecurityManager`, provided by Voyager, must be subclassed and specialized so that a security sandbox can be built to protect the local JVM and its applications. A customized `VoyagerSecurityManager` typically works with (if not installed by) the default policy to provide the sandbox. Methods on the `VoyagerSecurityManager` class determine whether classes are trusted or untrusted, that is, whether they have signers and whether or not static methods were invoked from a remote JVM instead of from local classes.

The method `VoyagerSecurityManager.isForeign()` determines whether checked method invocations were originated by code from a foreign system, or whether the method was invoked by a remote object.

When trusting classes based on their signer, the customized `VoyagerSecurityManager` must recognize signer objects and determine trust based on the policies of the currently installed `Policy` implementation. An array of classes on the stack are retrieved to determine whether any untrusted bytes are in the context; were any of the classes loaded by a classloader, and if so, do they all have trusted signers.

Many system-type method calls check whether a `VoyagerSecurityManager` is installed, and, if so, call the `VoyagerSecurityManager.checkMethodAccess( Class class, String method, Object arg )` method to see whether the calling code has the privilege to invoke this method. Any code loaded by the primordial classloader (from the classpath) should be trusted and allowed to invoke these methods. Otherwise, Voyager may be unable to start. Refer to Appendix A - Checked Methods for a list of all the methods that call `checkMethodAccess()`.

BasicPolicy Implementation

Overview

The BasicPolicy Voyager security policy implementation is a simple security policy that helps develop secure applications and serves as a reference for domain-specific policy development. Do not deploy production-quality applications with this implementation.

The BasicPolicy Implementation chapter details the following:

- Understanding the BasicPolicy architecture
- Installing BasicPolicy
- Securing a server with BasicPolicy
- Using the BasicPolicy library
- Using BasicPolicy with LDAP

Note: The BasicPolicy Implementation chapter assumes the reader is familiar with the Voyager Security Library and its architecture.

Understanding the BasicPolicy Architecture

The `BasicPolicy` implementation allows resources to be protected by simple username and password authentication and permission verification. All users, passwords and their granted permissions are organized in a simple access control list (ACL) stored in a Voyager directory server.

At startup, a copy of the ACL is retrieved and stored on the local virtual machine (VM). The ACL is managed by acquiring a reference to the `BasicPolicyManager`. Committing any changes made via the `commitChanges()` method places a copy of the new ACL instance in the remote directory.

The `BasicPolicy`, when initialized, can assume two modes: client and server. Initialized in client mode, the `BasicPolicy` acts as a factory for `UserPrincipals` and `UserSecurityContexts`. No authentication is performed in the client VM, nor should the client VM verify any permissions against the current user. The client is fully trusted to perform any task or use any resource available in the client VM.

Initialized in server mode, the `BasicPolicy` retrieves a copy of the ACL object from the remote directory server. Both user authentication and permission verification can be performed in the local VM. The `BasicPolicy` also installs an instance of `BasicSecurityManager` so that remote static method invocations and directory lookups are protected.

Installing BasicPolicy

Before using `BasicPolicy` to secure a Voyager application or host, initialize a remote directory with a simple ACL containing the administrative username and password. Create a simple text file declaring the users, their passwords and associated permissions. The `Installer` application uses this file to build an ACL object and store it in the target directory server.

The format of an entry in the ACL file follows:

```
grant username "username" password "password" { permission
com.foo.SomePermission "constraints", "actions"; };
```

Each `grant` entry creates a new user with the name `username` and the password `password`. An instance of the `permission` class specified after the permission keyword is constructed with the following text fields as constructor arguments.

```
New com.foo.SomePermission( "constraints", "actions" );
```

The `deny` entries are not supported by the `BasicPolicy`.

Create the following user roles with the listed permissions in the initial ACL file.

- `SystemPrincipal - com.recursionsw.security.policy.SystemPermission;`
- `SecurityOwner - com.recursionsw.security.policy.SecurityPermission;`

- DirectoryAdmin - com.recursionsw.security.policy.DirectoryPermission; and com.recursionsw.security.policy.SecurityPermission;

The only permission checked by the BasicPolicy and Voyager, by default, is the DirectoryPermission. DirectoryPermission allows for a secure directory server after the ACL is initialized in the directory. Subclass the BasicSecurityManager to restrict certain functions to the System or Security privileges.

After creating the ACL text file, use the com.recursionsw.security.admin.Installer application to initialize an ACL instance. Assuming an ACL text file named simple.acl was created, a remote directory server must be running to get started. At the command prompt, enter:

```
voyager 8000 -f 8000.db JNDI
```

To install the simple.acl, in another console window, enter:

```
java com.recursionsw.security.admin.Installer 8000/JNDI
simple.acl
```

Any existing instance of the ACL object is lost and replaced with the new ACL object installed.

Securing a Server with BasicPolicy

To secure a server is to specify the class name of a security policy implementation for use by the Voyager Security framework during startup of a Voyager server. The class name is retrieved from the System properties table. System properties can be set through a properties file using the -p option of the voyager.exe utility, or it can be set programmatically before Voyager.startup().

The property key and value for the BasicPolicy of the Voyager security provider is recursionsw.security.policy.1=com.recursionsw.security.policy.BasicPolicy, properties_file. The Voyager Security framework creates an instance of this class passing the file name following the comma to the constructor. BasicPolicy uses this second properties file to initialize itself.

Table 1 is a list of properties to include in the properties file read by BasicPolicy, with descriptions of each property.

Table 1 - BasicPolicy Properties

Policy	Description
policy.server	Must be set to true or false (default). If true, the BasicPolicy starts in server mode. If true, complete all subsequent fields.
directory.url	The Universal Resource Locator (URL) of the directory server. For example, //somehost.com:8000/JNDI
directory.username	The username the BasicPolicy uses to access the directory server. Grant this user DirectoryPermission.
directory.password	The password of the user that BasicPolicy uses to access the directory server.
security.owner.username	The username of the security owner.
security.owner.password	The password of the security owner.
security.system.username	The username the system uses to access the directory server. Grant this user DirectoryPermission. The principal associated to this username and password is used as the system principal.
security.system.password	The password the system uses to access the directory server. Grant this user DirectoryPermission. The principal associated to this username and password is used as the system principal.
security.administrator.username	The administrator username of the principal used to protect access to the PolicyManager implementation. If this value is null, the security owner principal is the default.
security.administrator.password	The password of the administrator.

Refer to Voyager Security Library for information on the security owner and the security administrator.

An example of a properties file for the `BasicPolicy` follows.

```
// if true, all subsequent fields must be filled in

policy.server=true

directory.url=7000/JNDI

directory.username=directoryAdmin
directory.password=directoryAdmin

security.owner.username=securityOwner

security.owner.password=securityOwner

security.system.username=systemPrincipal

security.system.password=systemPrincipal
```

Clients using servers secured by the Voyager Security provider `BasicPolicy` must also specify `BasicPolicy` as its security policy. The properties file is optional for the client and is only used to specify a security owner if needed. If a properties file is used by a client, `policy.server` equals `false`.

Using the BasicPolicy Library

The following sections describe the main classes in the `BasicPolicy` library.

UserPrincipal

`UserPrincipal` is the `BasicPolicy` implementation of the `java.security.Principal` interface. `UserPrincipal` embodies the identity of a particular entity, like the security owner role, or a user. `UserPrincipal` holds references to both the username and password for storage in the directory server.

The password is hashed in the constructor so that it cannot be read if an instance of `UserPrincipal` is serialized. If the requested hashing algorithm (SHA) is not installed on the local VM, the password remains obfuscated until it is deserialized into a VM with the requested hashing algorithm.

Note: Hashing is one way of rendering the password unreadable. Hashing cannot be reversed. Obfuscation can be reversed and should not be relied on for providing privacy.

Permissions

The `BasicPolicy` implementation comes with the following simple permission types in the `com.recursionsw.security.policy` package.

- `SecurityPermission`
- `InvokePermission`
- `SystemPermission`

- `DirectoryPermission`
- `AllPermission`

See the API Javadoc for details on these permissions.

All permissions used by the `BasicPolicy` implementation must implement both the `java.security.acl.Permission` interface and the `java.io.Serializable` interface.

Note: The `BasicPolicy` implementation restriction described above is the only restriction used in the BasicPolicy implementation; the Voyager security library enforces no restrictions on permission types.

BasicPolicyManager

The `BasicPolicyManager` is retrieved via the `BasicPolicy.getPolicyManager()` method. A reference to the security owner or security administrator principal must be passed to `BasicPolicy.getPolicyManager()`. Refer to Securing a Server with BasicPolicy and Voyager Security Library for information on the security owner and administrator roles.

If an administrator username is not supplied in the `BasicPolicy` startup properties file, the security owner protects the `BasicPolicyManager` instance.

After retrieving an instance, new users and permissions can be added to the ACL instance stored in the remote directory server. All the methods available on the `BasicPolicyManager` interface are described in the API Javadoc.

Note: After any changes are made, the `BasicPolicyManager.commitChanges()` method must be called to push a copy of the new ACL into the remote directory.

BasicSecurityManager

The `BasicSecurityManager` is installed only when the `BasicPolicy` is initialized in server mode. By default, the `BasicSecurityManager` only protects the internal directory by checking whether or not the calling entity has the `DirectoryPermission`.

`BasicSecurityManager` also protects specific Voyager API calls against remote static method invocations onto the local VM via the `checkMethodAccess()` method. The list of protected Voyager methods is in Appendix A - Checked Methods. All other overridden `SecurityManager` calls immediately fail when the current thread of execution is initiated by a remote VM via a remote method invocation.

UserSecurityContext

`UserSecurityContext` is an implementation of the security library `SecurityContext`. `UserSecurityContext` maintains the associations between the current thread of execution and the calling principal by wrapping the instance of `UserPrincipal` for which it must maintain the association.

Because the password is already hashed inside the `UserPrincipal`, the `UserSecurityContext` makes no additional efforts to protect its contents, apart from not offering any public accessors to the wrapped principal.

When the `UserSecurityContext` arrives in a remote VM, the wrapped `UserPrincipal` is re-authenticated with the local `BasicPolicy` implementation. If the re-authentication fails, a `SecurityException` is thrown to the calling VM, preventing the method from executing. The target method is responsible for verifying that the newly arrived `UserPrincipal` has the required permissions.

Using Basic Policy with LDAP

All of the `BasicPolicy` documentation applies when used with an Lightweight Directory Access Protocol (LDAP) directory server with the additions and notes described in the following sections.

Security

Currently Voyager only supports simple text string usernames and passwords for accessing LDAP directories.

BasicPolicy

`BasicPolicy` stores its access control list in a directory server using Java Naming and Directory Interface (JNDI). To use `BasicPolicy` with an LDAP directory server, add the following properties to the `BasicPolicy` properties file.

Table 2 - Properties Necessary To Use BasicPolicies With An LDAP Server

Property Key	Property Value
`recursionsw.directory.server.implementation`	`ldap`
`java.naming.factory.initial`	`com.recursionsw.ve.spi.jndi.VoyagerContextFactory`
`java.naming.security.authentication`	`simple`

The following is an example of a `BasicPolicy` properties file with the above properties added. In this example, Sun's LDAP service provider is used.

```
    // if true, all subsequent fields must be filled in
```

```
policy.server=true

directory.url=//localhost:7000/ACLDirectory

directory.username=directoryAdmin

directory.password=directoryAdmin

security.owner.username=securityOwner

security.owner.password=securityOwner

security.system.username=systemPrincipal

security.system.password=systemPrincipal

recursionsw.directory.server.implementation=ldap



java.naming.factory.initial=com.recursionsw.ve.spi.jndi.VoyagerCo
ntextFactory

java.naming.security.authentication=simple
```

`BasicPolicy` uses the `directory.username` and the `directory.password` properties as the username and password to access the directory server. The user must have read and write access to the Voyager directory tree. Also the `directory.url` entry is the URL used by the `BasicPolicy` to access the directory. In this example, `//localhost:7000` is the URL to the directory server, and the Voyager directory tree name is `ACLDirectory`.

To use an initial `ContextFactory` other than Sun's, use the `recursionsw.alternate.factory.initial` property.

Installer

`aclinst` is a command line tool used to load a `BasicPolicy` ACL into the directory server. To use `aclinst` with an LDAP directory server, pass a properties file as the second parameter when invoking `aclinst`. For example:

```
aclinst //dallas:389/ACL simple.acl ldap.props
```

The above example specifies a directory server on a machine named `dallas` listening on port `389`. The Voyager directory tree name is `ACL`.

The properties file must contain the properties specified in **Table 3**.

Table 3 - Required Properties File Contents

Property Key	Property Value
recursionsw.directory.server.implementation	ldap
java.naming.factory.initial	com.recursionsw.ve.spi.jndi.VoyagerContextFactory
java.naming.security.authentication	simple
java.naming.security.principal	A simple text string username used to log into the directory server.
java.naming.security.credential	A simple text string password used to log into the directory server.

The following is an example properties file.

```
java.naming.factory.initial=com.recursionsw.ve.spi.jndi.VoyagerCo
ntextFactory

java.naming.security.authentication=simple
java.naming.security.principal=directoryUser

java.naming.security.credential=userPassword

recursionsw.directory.server.implementation=ldap
```

The example properties file uses Sun's service provider. The user represented by the `java.naming.security.principal` and `java.naming.security.credential` entries must have read and write access to the Voyager directory tree. To use an initial `ContextFactory` other than Sun's, use the `recursionsw.alternate.factory.initial` property.

Communication Security

Traversing Firewalls

Overview

Voyager can be configured to traverse firewalls to reliably and safely use Voyager on the internet.

The Traversing Firewalls chapter details the following:

- Tunneling through SOCKS and Hypertext Transfer Protocol (HTTP) firewalls
- Using HTTP and SOCKS 4 to tunnel from the Light Client
- Installing and using SOCKS 4 for tunneling
- Using SOCKS 5 for tunneling
- Using HTTP for tunneling

Note: The Traversing Firewalls chapter assumes that the reader is familiar with SOCKS 4 or 5 and HTTP tunneling.

Tunnelling Through Firewalls

Two common protocols are used by applications to traverse a firewall, HTTP tunneling and SOCKS tunneling.

HTTP tunneling is based on an extension of the HTTP protocol and has become very popular because many organizations already have HTTP proxy servers installed. One of the major drawbacks of HTTP tunneling is the simple (and potentially insecure) authentication mechanism used to validate a user's identity.

SOCKS is also a commonly used standard. Two versions are available, SOCKS versions 4 and 5. SOCKS 4 does not support true authentication; only a username is passed between the client and firewall. SOCKS 5 is more robust and is defined in Request For Comment (RFC) 1928. Note that a few authentication mechanisms are required by the specification, such as RFC 1919 "Username/Password Authentication." The specification also defines mechanisms to allow additional authentication be used by the SOCKS 5 implementation.

Voyager can be configured to traverse firewalls programmatically using Voyager Security Application Program Interfaces (APIs)

In Voyager, socket connections are made based on socket policies. A socket policy defines the type of connection and the parameters required to make a successful connection to the end point. Tunneling socket policies defined by Voyager are listed in **Table 1**.

Table 1 - Voyager Tunneling Socket Policies

Name	Class Name	Properties	Use
TCP (Default)	`com.recursionsw.ve.transport.impl.tcp.TCPSocketPolicy`	Timeout	Used for normal socket connection to end point.
SOCKS 4	`com.recursionsw.ve.transport.impl.tcp.socks.Socks4SocketPolicy`	Timeout Socks Proxy server name Socks Proxy server port Username	Used for SOCKS4 tunnel connection to end point.
SOCKS 5	`com.recursionsw.ve.transport.impl.tcp.socks.Socks5SocketPolicy`	Timeout Socks Proxy server name Socks Proxy server port Username User password	Used for SOCKS5 tunnel connection to end point.
HTTP	`com.recursionsw.ve.transport.impl.tcp.http.HtpTunnelSocketPolicy`	Timeout HTTP Proxy server name HTTP Proxy server port Userame User password	Used for HTTP tunnel connection to end point.

By default, Voyager uses the standard Transmission Control Protocol (TCP) socket policy (the policy for a normal TCP/IP socket connection). Voyager clients should use an HTTP or SOCKS socket policy to make an HTTP or SOCKS tunnel connection to a server. The client can register the Internet Protocol (IP) address range and the corresponding socket policies to be used by the `SocketPolicyManager`.

For example, consider the register of range-to-socket policy mappings in **Table 2**.

Table 2 - Range-to-Socket Policy Mappings

Range	Socket Policy
10.2.3.8:80	SOCKS 5
10.2.3.*:80	HTTP
10.2.*: -	TCP

Given the mappings in **Table 2**,

- A socket connection to `10.2.3.8:80` uses SOCKS 5
- A socket connection to `10.2.3.5:80` uses HTTP
- A socket connection to `10.2.1.5:80` uses TCP

Note the following:

- The user specifies a socket policy for an endpoint.
- By default, Voyager uses the TCP socket policy.
- The socket policy to be used for a given end point can be programmatically specified using the `ManagerTcpPolicy`..
- The endpoint is specified as a `HostAddressRange`. Refer to HostAddressRange for details.

HTTP Tunneling Example

Using the following parameters and code, an HTTP Tunnel Connection can be established between a Voyager client and a Voyager Server. For the following example, the Voyager server is behind a firewall. The Voyager server has the following:

- Machine name = `server.recursionsw.com`
- Port = `9000`

The firewall is an HTTP Proxy Server with the following:

- Machine name = `proxy.recursionsw.com`
- Port = `8080`

- Username = `admin`
- Password = `pass`

To specify the socket policy used to connect to the Voyager server described above, from a Voyager client, use the following sample client code.

```
ManagedTcpTransport.enable( true );

ManagedTcpPolicy managedTcpPolicy = new ManagedTcpPolicy( null, 1 );

SocketPolicyList policyList =
managedTcpPolicy.getServerSocketPolicyList();



HostAddressRange range = new
HostAddressRange( "server.recursionsw.com:9000" );

HttpTunnelSocketPolicy policy = new HttpTunnelSocketPolicy( "admin",
"pass", "proxy.recursionsw.com", 8080 );

policyList.setSocketPolicy( policy, range );



ManagedTcpTransport transport = (ManagedTcpTransport)
TcpTransport.getTransport();

transport.setPolicy( managedTcpPolicy );
```

To tunnel through a SOCKS firewall, use the SOCKS 5 socket policy in a similar manner to that described above.

Tunneling With SOCKS 4

TBD

Tunneling With SOCKS 5

The SOCKS 5 tunneling policy class is `com.recursionsw.ve.transport.impl.tcp.socks.Socks5SocketPolicy`. The configuration is identical to SOCKS 4 with the additional requirement of a password.

Tunneling with HTTP

The HTTP tunneling policy class is `com.recursionsw.ve.transport.impl.tcp.http.HttpTunnelSocketPolicy`. The configuration information required is the same as for SOCKS 5.

Encrypted Connections

Voyager supports both clear text and encrypted connections between client and server contexts. While Voyager provides an API for specifying encrypted connections, the declarative approach is the simpler and strongly preferred approach.

Voyager's Java SE secure sockets policy implementation, realized by `com.recursionsw.ve.transport.impl.tcp.ssl.JSSEClientSocketPolicy`, for client sockets and `com.recursionsw.ve.transport.impl.tcp.ssl.JSSEServerSocketPolicy` for server sockets, is implemented on Java Secure Socket Extension (JSSE).

Voyager's .NET client side and server side secured socket policy implementations are realized by recursionsw.voyager.security.ssl.SSLClientSocketPolicy and recursionsw.voyager.security.ssl.SSLServerSocketPolicy respectively.

The policy's HostAddressRange object specifies which connections are encrypted. Direct use of the JSSE policies for Java SE or SSL policy for .NET requires careful coding, and requires careful handling of the values used to initialize JSSE. Construction of the HostAddressRange object specific to every context requires understanding the network at the level of IP addresses, with the associated configuration management issues, leads to maintenance issues in all the but simple environments.

As an alternative to working directly with `JSSEClientSocketPolicy` / `JSSEServerSocketPolicy for Java SE or` SSLClientSocketPolicy/ SSLServerSocketPolicy for .NET, policies can be specified by name at the level of client and server contexts. This preferred approach is based on Voyager properties whose name begins with the context name and whose values configure the connection policy.

Endpoint Policies

Endpoint policies are managed by and created as needed by the `EndpointPolicyFactory`. A reference to the singleton `EndpointPolicyFactory` is available from the `ContextManager`. This factory produces an implementation of `IEndpointConfigurationPolicy` for every client or server context. Voyager automatically installs in `EndpointPolicyFactory` a default policy that causes a context's endpoints to create unencrypted connections. Voyager also ships with an implementation of `IEndpointConfigurationPolicy` that, when added to `EndpointPolicyFactory` and configured, will cause endpoints of a context having the context name specified in the policy to use encrypted connections. Because the `IEndpointConfigurationPolicy` is specified at the level of a context rather than by IP address, management of secure connections becomes easier than directly coding the socket policies.

Endpoint Policy Factory

The `EndpointPolicyFactory` configures new instances of `IEndpointConfigurationPolicy` using values retrieved from Voyager's collection of properties. When a client or server context needs to open an endpoint the context asks the `EndpointPolicyFactory` for an `IEndpointConfigurationPolicy` object configured for the context if it doesn't already have one. The context then asks the policy to create, based on the endpoint's connection URL, and apply the appropriate socket policy. Endpoint configuration policies are added to `EndpointPolicyFactory` programmatically by calling `registerEndpointConfigurationPolicy()` or by a line in a Voyager properties file.

Endpoint Configuration Policies

Implementations of `IEndpointConfigurationPolicy` are responsible for creating and applying a connection policy. Each `IEndpointConfigurationPolicy` instance is configured for a single context name and applies only to endpoints created by a single context.

The default implementation, `ClearTextEndpointConfigurationPolicy`, is loaded when Voyager starts. The policies it creates and applies cause the socket factories to produce unencrypted connections.

In Java SE the `JSSEEndpointConfigurationPolicy` implementation is loaded only when explicitly requested. The policies `JSSEEndpointConfigurationPolicy` creates and applies cause the socket factories to produce connections encrypted using the configured provider and protocol, e.g., protocol "TLS" or "SSL" and provider "SunJSSE".

In .NET the `SSLEndpointConfigurationPolicy` implementation is loaded only when explicitly requested. The policies `SSLEndpointConfigurationPolicy` creates and applies cause the socket factories to produce connections wrapped by SslStream. It provides a stream used for client-server communication that utilizes the Secure Socket Layer (SSL) security protocol to authenticate the server and the client.

Additional endpoint configuration policies can be added by implementing `IEndpointConfigurationPolicy` and registering the implementation with `EndpointPolicyFactory`.

Installing an Endpoint Configuration Policy

An endpoint configuration policy other than the default policy is installed by adding it to the `EndpointPolicyFactory`. The normal mechanism is to include in the Voyager property file one line for each additional policy. The line must begin with `Voyager.messageprotocol.EndpointPolicyFactory.addEndpointPolicy` followed by an equal sign followed by the name of the implementation class. The implementation class must implement `IEndpointConfigurationPolicy`. Here's an example.

```
Voyager.messageprotocol.EndpointPolicyFactory.addEndpointPolicy=JSSEPolicy
```

The above line uses a built-in abbreviation for the JSSE policy. Here is the illstrustration of the corresponding .NET counter part

```
Voyager.messageprotocol.EndpointPolicyFactory.addEndpointPolicy=SSLPolicy
```

For a user-provided policy the right side of the property must be a fully qualified class name, such as the following line. (In the property file this must be a single line.)

```
Voyager.messageprotocol.EndpointPolicyFactory.addEndpointPolicy=
com.recursionsw.ve.messageprotocol.JSSEEndpointConfigurationPolicy
```

Configuring the Default Policy

The name of the default policy implementation is `RSIcleartext`. This policy requires no configuration.

Configuring the JSSE Policy for Java SE

The name of the Voyager policy implementing secure connections using JSSE is `RSIjsse`. A line in the Voyager property file, such as the one seen below, associates a context name with the JSSE endpoint policy. The example line declares that a client or server context named `securedserver` will use the Voyager implementation of the JSSE endpoint configuration policy. The string `.endpointconfigpolicy`, in lower case letters, must be appended to the context name to mark this as the property value for endpoint configuration.

```
securedserver.endpointconfigpolicy=RSIJSSE
```

The right side of the above line is the name of the policy as returned by `IEndpointConfigurationPolicy`'s `getName()` method, and is compared with the name from the property without regard to letter case.

Instances of the policy require the configuration arguments listed in Table 1. Note that every property name starts with the name of the context to which the value applies, and that the entire property name is case sensitive. Most of the property values are also case sensitive.

Table 1: JSSE Configuration Properties

Property Name	Parameter Description	Example Value
<context name>.PassPhrase	The passphrase provided to the key store and key manager factory.	passphrase
<context name>.SSLContext	The name of the protocol that is passed to `SSLContext.GetInstance()`	TLS
<context name>.KeyManagerFactory	The name of the algorithm passed to the key manager factory.	SunX509
<context name>.TrustManagerFactory	The name of the algorithm passed to the trust manager factory.	SunX509
<context name>.KeyStore	The type of key store.	pkcs12
<context name>.KeyFile	The file used to build a stream that is passéd to the keystore, i.e., the file that contains the keys.	examples/jsse/testkeys

Configuring the SSL Policy for .NET

The name of the Voyager policy implementing secure connections using SSL is `RSIssl`. A line in the Voyager property file, such as the one seen below, associates a context name with the SSL endpoint policy. The example line declares that a client or server context named `securedserver` will use the Voyager implementation of the SSL endpoint configuration policy. The string `.endpointconfigpolicy`, in lower case letters, must be appended to the context name to mark this as the property value for endpoint configuration.

```
securedserver.endpointconfigpolicy=RSISSL
```

The right side of the above line is the name of the policy as returned by `IEndpointConfigurationPolicy`'s `getName()` method, and is compared with the name from the property without regard to letter case.

Instances of the policy require the configuration arguments listed in Table 2. Note that every property name starts with the name of the context to which the value applies, and that the entire property name is case sensitive. Most of the property values are also case sensitive.

Table 2: SSL Configuration Properties

Property Name	Parameter Description	Example Value
<context name>.CertificateFilePath	The CertificateFilePath is the path to the server certificate	passphrase
<context name>.ServerName	The name of the machine that has server certificate installed	johndoe-xp
<context name>.ServerCertificateName	The name of the server certificate	johndoe-xp
<context name>.NeedClientAuth	Indicating client need to validate server certificate	Required or none

Appendix A: Checked Methods

Overview

Appendix enumerates all the Voyager Library classes and methods that call `VoyagerSecurityManager.checkMethodAccess( Class type, String method, Object arg )` to verify that the calling code or principal has the required permissions to invoke it. Some methods will provide an argument value that may affect the trust decision.

The `checkMethodAccess()` method is only called when a `VoyagerSecurityManager` subclass is registered with the `java.net.System` class.

For example, the first line in `ClassManager.setParentClassLoader()` tests to see whether a `java.net.SecurityManager` subclass is installed. If so, if it is an instance of `VoyagerSecurityManager`, the method `checkMethodAccess( ClassManager.class,` "setParentClassLoader", null) is invoked. If the VoyagerClassLoader instance deems the calling principal untrusted, a SecurityException is thrown.

Checked Methods

Table 1 enumerates all the Voyager-checked methods and the argument types if applicable. Descriptions are only provided where information beyond the **Voyager Core Developer's Guide** and **Voyager** API will help describe what is being protected. Caveats to consider are also noted.

Table 1 – Voyager-Checked Methods

Class Name	Method	Argument	Description
com.recursionsw.ve.ClassManager	setParentClassLoader		
com.recursionsw.ve.ClassManager	resetClassLoader		Protects the current VoyagerClassLoader from being recycled, and forcing the reloading of all application classes
com.recursionsw.ve.ClassManager	enableResourceServer		
com.recursionsw.ve.Namespace	register		
com.recursionsw.ve.Namespace	deregister		
com.recursionsw.ve.Namespace	setDefaultProtocol		
com.recursionsw.ve.ThreadManager	setMaxIdleThreads		
com.recursionsw.ve.Voyager	getSystemThreadGroup		
com.recursionsw.ve.Voyager	shutdown		
com.recursionsw.ve.Voyager	registerService		
com.recursionsw.ve.Voyager	deregisterService		
com.recursionsw.ve.Voyager	addSystemListener		
com.recursionsw.ve.Voyager	removeSystemListener		
com.recursionsw.ve.Voyager	enableRMIServer		
	registerAdapter		
	getAdapter		
	getDefaultAdapter		
	getContext		
	getCurrentOrNull		
	copyContext		
	addContext		
	removeContext		
	put	key	
	add	key	
	get	key	
	getAllEntries	key	

	getObjectEntries	key	
	getDirectoryEntries	key	
	remove	key	
com.recursionsw.ve.directory.IDirectory	clear		
com.recursionsw.ve.directory.IDirectory	getValues		
com.recursionsw.ve.directory.IDirectory	getKeys		
com.recursionsw.ve.loader.VoyagerClassLoader	addResourceLoader	priority	Protects code from adding resources at a specific priority
com.recursionsw.ve.loader.VoyagerClassLoader	removeResourceLoader		
com.recursionsw.ve.loader.VoyagerClassLoader	setResourceLoadingEnabled		
com.recursionsw.ve.loader.VoyagerClassLoader	getLocalResourceAsStream		
com.recursionsw.ve.message.IInvoker	invoke	classname	Protects static method invocations on given class name
com.recursionsw.ve.message.IInvoker	construct	classname	Protects instance construction of the given class name
com.recursionsw.ve.security.Policy	runAs		Protects having the security context associated with the current thread removed
com.recursionsw.ve.security.Security	getPolicyNames		
com.recursionsw.ve.security.Security	addPolicy		
com.recursionsw.ve.security.Security	setSystemPrincipal		
com.recursionsw.ve.security.Security	getSystemPrincipal		
com.recursionsw.ve.security.Security	defaultReceiveAs		
com.recursionsw.ve.security.Security	defaultInvokeAs		
com.recursionsw.ve.security.Security	getCurrentPrincipal		
com.recursionsw.ve.transport.impl.t cp.TcpTransport	setServerListenBacklog		
com.recursionsw.ve.transport.Transport	register		
com.recursionsw.ve.transport.Transport	setDefaultTransport		
com.recursionsw.ve.transport.Transport	addRequestHandler		
com.recursionsw.ve.messageprotocol.vr mp.VrmpReference	setPropagateContext		Protects code from disabling context propagation

Appendix B: SSL Examples

The SSL examples demonstrate how to configure Voyager to use secure sockets for client and server connections.

Note that the shipped examples use a self-signed certificate and will fail in some cross platform configurations because the client does not recognize the certificate authority. The usual solution is to replace in the examples certificates (with the associated keys and, if needed, the intermediate certificates) provided by a recognized certificate authority.

JSS Example

The `JSSClient and JSSServer` programs demonstrate using the Network Security Services for Java (JSS) as a secure socket provider for Voyager client and server connections. This example also serves to show how to implement custom socket factories for use with Voyager, since JSS support is not bundled with the security jar file.

Please note as of May 2011 jss4.dll only works on 32-bit environment. Since the JSS Example relies on jss4.dll. It only works on a 32-bit platform.

JSSE Example

The `JSSEClient and JSSEServer` programs demonstrate using the Java Secure Socket Extension (JSSE) as a secure socket provider for Voyager client and server connections. The custom socket factories for JSSE are bundled with the security jar file. The `JSSEDeclarativeClient` and `JSSEDeclarativeServer` programs demonstrate declaratively configuring secure connections.

Security Source code location for java

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc\java\examples\ directory. The JSS examples are in the jss\ directory, and JSSE examples are under the jsse\ directory, and declarative JSSE examples are under the declarejsse\ directory.

Security Source code location for .net

The examples are located at %VOYAGER_HOME%\examples\csharp\window-dontnet\security directory.