Basic DOF Security

Programmer's Guide

Version 7.0

Table of Contents

Chapter 1: Introduction

The *Basic DOF Security Programmer's Guide* contains the knowledge system developers need to use version 6.x of the DOF Object Access Libraries (OALs) to connect DOF nodes in secure domains. This guide does not cover advanced security topics, such as setting up an Authentication Server (AS) and domain database storage. This guide is intended as a practical guide to implementing security in code.

The DOF OALs are available in Java, C, and C#. This guide provides information only for using the Java OAL.

Before reading this guide, you should be familiar with the material in the *DOF Connectivity Programmer's Guide*. Another useful reference is the API documentation, the latest version of which is included in the SDK.

How to Read This Guide

To get the most from this guide, you should read each chapter in order through to the end.

This guide can be used in conjunction with the security training in the DOF Essentials SDK. The training packages the full source code from which the examples in this guide were extracted. Refer to it for more context for any of the samples in this guide. The training code may be copied into your own development environment and used for practice. In addition, while this guide provides samples only in Java, the security training in the SDK also includes samples for the C and C# OALs.

Security Concepts Overview

DOF centralizes security information on a special node called the Authentication Server (AS). Trust is established between peers based on a shared trust of the AS. The peers know their own credentials (identity and secret), and although they will learn other peers' identities, they never learn their secrets.

The AS is the only node that knows all the security information for a domain and all of the permissions that have been granted to each node. Centralizing security information on an AS allows requestors and providers to act as peers in secure communications, rather than assuming that one node is more powerful or knowledgeable than the other. Requestors and providers can then be distributed throughout a network.

Note: DOF users familiar with KryptoKnight may find it useful to know that DOF security is based on this technology.

Each AS is connected to one or more data repositories, called a “domain storage.” Each repository contains the security information for a single security domain. Although communication can be set up between nodes in different security domains, in the beginning, it is useful to think of DOF network traffic as being segmented by security domains. In other words, only nodes in the same domain can communicate.

Authentication in a domain is achieved through the credentials mentioned earlier, which consist of an identity and secret. The operations a node is able to perform within the domain are controlled through permissions.

Although this discussion has referred to communication between nodes, it is important to note that DOF security is actually more granular than that. Credentials are actually assigned to DOF connectivity components (DOFSystem, DOFConnection, and DOFServer). Thus, it is possible for a DOF to have a DOFSystem with credentials in one domain and a DOFConnection with credentials in a different domain. These components would then be unable to communicate with one another without bridging them, even though they are attached to the same DOF. However, in the most common use cases, all the connectivity components attached to the same DOF would use the same credentials, so it is a useful simplification in the beginning to think of security in terms of communication between nodes.

Roles

For the purposes of this guide, there are three types of developers:

- System developers
- Application and device developers (operations programming)
- Domain managers, who ensure data is properly entered into the domain storage repository

This guide was written specifically for system developers, those who will be working with the connectivity functionality of DOF and providing classes for application and device developers (operations programming).

For more information on these roles, refer to the *DOF Connectivity Programmer’s Guide*.

Chapter 2: Authentication Server Access

To test or use secure connectivity components, you must have access to an Authentication Server (AS) that can authenticate the components' credentials. This chapter explains how to install and connect to the AS that is included with the security training.

Installing and Starting the AS

The AS can be installed on either Windows or Linux. Follow the instructions in the readme files for the tutorials to install and run the trainings. This starts the AS as well as other sample applications. You may then close sample applications, but leave the AS running, if you choose.

The chapters in this guide correspond with the following training modules:

- *Chapter 3: DOFConnection and DOFServer Instances* with training module 1
- *Chapter 4: DOFSystem Instances* with training module 2
- *Chapter 5: Permissions* with training module 3
- *Chapter 6: Bridging Security Domains* with training module 4

The AS created for training module 3 (on permissions) contains limited permissions for the purpose of demonstrating exceptions in training, so using that AS without understanding the permissions it grants may lead to unexpected results. You may wish to use the AS from an earlier training module while experimenting with code in *Chapter 5: Permissions*.

Warning: The training AS should be used only for training or experimentation while learning DOF, and it should be run only within an isolated environment. It must never be used in development or in a final production system. It provides very broad permissions with weak secrets. In addition, because it is distributed to multiple audiences, it violates the general DOF specification to create unique object identifiers (OIDs).

Connecting to the AS

Connections to the AS are unsecured. For this reason, they should allow only authentication-related traffic, and operations should not be sent over these connections. All authentication-related traffic is encrypted.

Code Sample

The following sample shows a sample configuration for a connection to an AS:

```
DOFConnection.Config connectionConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, addressOfAS)
   .addTrustedDomains(DOFObjectID.DOMAIN_BROADCAST)
   .setProtocolNegotiator(DOFProtocolNegotiator.createDefaultASOnly())
```

```
    .build();
```

Code Discussion

You should already be familiar with the builder pattern and how it is used to create DOF connectivity components. The addressOfAS variable passed in the Builder's constructor represents a DOFAddress instance that is using the correct IP address and port of the AS. You need to create your own DOFAddress using the IP address and port of the node where you installed the AS (for training or experimentation, this may often be the localhost address). For help with the builder pattern or with creating a DOFAddress instance, refer to the *DOF Connectivity Programmer's Guide*.

The methods used in the sample do the following:

- The *addTrustedDomains* method specifies which domains can be authenticated over the connection and creates a route to the AS for these domains. You can pass a list of specific domains in this method to limit which domains the connection attempts to connect to; however, the sample above uses the domain broadcast OID, which means it will attempt to find any domain. This is sufficient for experimentation and testing, but in production systems, it is probably more efficient to provide a list of specific domains.
- The *setProtocolNegotiator(DOFProtocolNegotiator.createDefaultASOnly())* method and argument cause the connection to be used only for traffic associated with validating credentials and granting permissions. This is useful for strengthening security on connections to the AS, because although the connection itself is unsecured, regular operations cannot be sent over it. It can be used only for authentication.

If you are setting up a testing environment, you can create a connection to the AS from multiple nodes; however, it is probably simplest to attach the AS connection to a DOF that serves as a proxy, add a secure server (discussed in the next chapter), and have all other nodes connect to the proxy.

Chapter 3: DOFConnection and DOFServer Instances

Creating secure connections and servers is a matter of creating credentials, creating the components, and setting credentials in the components' configurations. You also need to understand the effect of setting the security desire in the configuration.

Creating Credentials

Because secure connectivity components must be authenticated, they require credentials. A set of credentials is made up of the following:

- **Domain Identifiers.** Each domain has an identifier. Domain identifiers are not required in a set of credentials, but it is strongly recommended that you include them if they are known. The domain identifier must be an object identifier (OID) that is properly constructed according to DOF rules for creating OIDs. The class for domain identifiers is called DOFObjectID.Domain. It is instantiated using a *create* method.
- **Authentication Identifiers.** The authentication identifier is comparable to a username for the node—it represents the component's identity. It must be a properly constructed OID. The class for authentication identifiers is called DOFObjectID.Authentication. It is instantiated using a *create* method.
- **Secrets.** Each authentication identifier is associated with a secret, such as a password or private 256-bit key, that is shared between the node and the AS. A password is instantiated as a String, and a key is a byte array.

The Authentication Server (AS) included with the security training has specific sets of credentials in its domain storage, so to create credentials that work with that AS, you must use the exact OIDs and keys shown in the sample code. It includes separate credentials for the provider, requestor, and proxy nodes.

Code Sample

To create credentials, use lines of code such as the following:

```
String domainOID = "[6:tech-services.opendof.org]";
DOFObjectID.Domain domain = DOFObjectID.Domain.create(domainOID);

String proxyOID = "[3:proxy@tech-services.opendof.org]";
DOFObjectID.Authentication proxyAuthentication =
         DOFObjectID.Authentication.create(proxyOID);

byte[] key = DOFUtil.hexStringToByteArray
         ("0000000000000000000000000000000000000000000000000000000000000000");

DOFCredentials proxyCredentials = DOFCredentials.Key.create
         (domain, proxyAuthentication, key);
```

Code Discussion

The preceding sample creates the credentials associated with a proxy node. Create additional credentials for a requestor and provider using the following OIDs with the same key:

- [3:provider@tech-services.opendof.org]
- [3:requestor@tech-services.opendof.org]

You may place credentials in their own class and make them static, so they can be easily used by any class in the package. See the Creds class in the sample security training for an example.

Note: Credentials are always passed by reference and never cloned.

Sample code in this guide uses the variable names shown in the preceding sample. It uses providerCredentials and requestorCredentials as the variable names for the DOFCredentials associated with providers and requestors, respectively.

Security Desire

The SecurityDesire enumeration enables you to control the types of security allowed on DOFConnections and DOFServers. The OALs provides the following useful values:

- **NOT_SECURE.** This value prevents the component from validating credentials and should not be used in a secure system.
- **ANY.** If the security desire for a connection that has credentials is set to ANY, it can connect to a server either with or without validating its credentials. A server that has credentials and has its security desire set to ANY can accept either secure or unsecured connections. The default security desire for both connections and servers is ANY.
- **SECURE.** If the security desire for a connection or server is set to SECURE, connections cannot be made without validation.

The SECURE and ANY security desires are compatible. If a connection set to SECURE attempts to connect to a server set to ANY, the connection will succeed as long as both components have compatible credentials. The reverse is also true, and a secure server can accept a connection with the security desire set to ANY, as long as both have credentials.

Note: If both components have their security desire left at the default security desire of ANY, no validation will occur, even if both components have credentials. *To ensure that a secure connection is made, either one or both components must set its security desire to SECURE.*

DOFConnection Instances

The following sample shows how to build a configuration for a secure DOFConnection.

Code Sample

```
DOFConnection.Config myConnConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, myAddress)
   .setCredentials(requestorCredentials)
   .setSecurityDesire(SecurityDesire.SECURE)
   .build();
```

Code Discussion

The configuration shown uses the DOFCredentials created earlier for a requestor node. The security desire is set to SECURE so that the connection will always require validation.

DOFServer Instances

The following sample shows how to build a configuration for a DOFServer that can accept secure or unsecured connections, using the DOFCredentials created earlier for a proxy node.

Code Sample

```
DOFServer.Config myServerConfig = new DOFServer.Config.Builder
         (DOFServer.Type.STREAM, myAddress)
   .addCredentials(proxyCredentials)
   .setSecurityDesire(SecurityDesire.ANY)
   .build();
```

Code Discussion

The security desire is shown set to ANY for reference, although this step is unnecessary since ANY is the default value.

Because the security desire for the connection shown earlier is set to SECURE, this server can accept the connection, and validation occurs. In addition, this server could accept connections in the unsecured domain that do not have credentials. It would also accept connections with credentials that have a security desire of ANY, but communication between those components would also be in the unsecured domain. The connection would be unable to communicate with any other secure components, even if they had compatible credentials, because validation would not occur.

Exceptions

The Java OAL defines several security-related exceptions, two of which you may receive at this point if you are attempting to validate and connect secure components:

- Authentication failed
- Unknown domain

Authentication Failed

The “authentication failed” exception may mean your credentials are incorrect in some way, although this is a general exception for a number of security-related failures.

Unknown Domain

The “unknown domain” exception typically means the domain identifier used in your credentials is incorrect or the route to the AS has been lost. If it is a connection failure, the failure may have occurred in any of the connections between the node and the AS.

Chapter 4: DOFSystem Instances

When creating a secure DOFSystem, it is important to first ensure that the DOF has a compatible route to the Authentication Server (AS); otherwise, the system will not be able to authenticate and creation will fail. The library provides a couple of different methods for checking that a domain is available, which are suitable for different use cases. Both involve the creation of a DOFDomain and are discussed in the next section. After looking at these, we can proceed with creating a secure DOFSystem.

The DOFDomain Class

The DOFDomain class represents the route to a domain and enables checking the state of domain connectivity. This class can be used in a couple of ways to ensure that a route to the domain is available when you attempt to instantiate a DOFSystem:

- Domain configurations (DOFDomain.Config instances) can be added to DOFConnection and DOFServer by using the *addDomains* method in their builders. This ensures that these components check the route to each of the added domains when they connect or start. The connection fails if routes are not available to all specified domains. In addition, it causes the connection or server to periodically check that the domain is still available. If the route to the domain fails, the connection is disconnected. Thus, if the connection or server is connected, you can assume that the route to the domain is open. This use of DOFDomain is most useful where connections are being made synchronously.
- An independent instance of DOFDomain can be instantiated, and a DOFDomain.StateListener can be used to monitor the route to the domain. The DOFSystem itself can be created in the listener callback to ensure that you attempt to create an instance of DOFSystem only when a route to the domain is available. This use of DOFDomain is effective with asynchronous connections.

Code Sample

DOFDomain.Config is instantiated through the use of a builder:

```
DOFDomain.Config domainConfig = new DOFDomain.Config.Builder(proxyCredentials)
   .setMaxSilence(30 * 60 * 1000)
   .build();
```

Code Discussion

The builder requires a DOFCredentials argument, and in this sample, we used the same credentials used for other components on the node. In addition, the *setMaxSilence* method controls how often (in milliseconds) the route to the domain is checked to ensure it remains open. The default setting is for one hour, but if your application frequently creates and destroys DOFSystems, you may wish to lower this considerably. The sample sets the maximum silence to 30 minutes, although this may still be too high for some applications.

Secure DOFSystem Instances

In this section, we will walk through the two different use cases for DOFDomain when instantiating a DOFSystem.

Adding a Domain Configuration to a Connection

The high-level steps for this use case are the following:

1. Build a connection configuration that adds the domain and connect synchronously.
2. Instantiate DOFSystem.

Build a Connection Configuration That Adds the Domain

The following sample shows the entire process of building a connection configuration, creating the connection, and connecting synchronously.

Code Sample

```
DOFConnection.Config connectionConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, address)
   .setCredentials(proxyCredentials)
   .addDomains(domainConfig)
   .setSecurityDesire(SecurityDesire.SECURE)
   .build();

DOFConnection connection = dof.createConnection(connectionConfig);

try{
   myConnection.connect(TIMEOUT);
} catch (DOFException e){
   System.out.println("Unable to connect. There may not be a valid route to
            the AS.");
   e.printStackTrace();
}
```

Code Discussion

You should be familiar with most of the code shown in the sample, but note that the sample shows the use of the *addDomains* method in the builder.

Instantiate DOFSystem

The following sample shows how to create a secure DOFSystem.

Code Sample

```
DOFSystem.Config systemConfig = new DOFSystem.Config.Builder()
   .setCredentials(proxyCredentials)
   .build();

try {
   DOFSystem system = dof.createSystem(systemConfig, TIMEOUT);
} catch (DOFException e) {
   e.printStackTrace();
}
```

Code Discussion

To create a secure DOFSystem, you must build a system configuration (unlike the method for creating a default system that was shown in the *DOF Connectivity Programmer's Guide*). Because the system must now be authenticated when it is created, you must place the *createSystem* method in a try-catch block and provide a timeout argument.

Using a Domain State Listener

A domain state listener can be implemented in multiple ways. This guide illustrates a single, highly specific example. The high-level steps for this example are the following:

1. As the system developer, you would do the following:
   a. Create an interface with a method that can be used to pass a DOFSystem to the application developer when it becomes available.
   b. Implement DOFDomain.StateListener and instantiate DOFSystem in its *stateChanged* method. Also call the interface method from step 1.1 in the *stateChanged* method to pass the DOFSystem to the application developer.
   c. Write a method that the application developer can call to initiate system creation.
2. The application developer would use your code by doing the following:
   a. Implementing the interface you created in Step 1.1.
   b. Calling the method you created in Step 1.3.

Both system and application developer steps are described in the following sections.

Create System Available Interface

The callback in this interface will be called in the *DOFDomain.StateListener.stateChanged* method when it attempts to create a DOFSystem.

Code Sample

```
public interface SystemAvailableCallback{
   public void systemAvailable(String name, DOFSystem system, Exception
            exception);
}
```

Code Discussion

The main purpose for creating an interface is so that you can provide application developers with code that includes the interface, and they can then implement the interface to create one or more systems, as needed. For this reason, our sample interface includes a parameter for the system name, which is used as a key for mapping multiple systems. The other two parameters enable the callback to pass either a created DOFSystem to the application developer or an exception if the system could not be created.

Implement DOFDomain.StateListener

The DOFDomain.StateListener interface has two methods:

- **stateChanged.** The library calls this method when the connected state of the domain changes. It is also called when the DOFDomain is instantiated to provide the initial state of the domain.
- **removed.** The library calls this method when the listener is removed or the system is destroyed. You can use it to free resources associated with the listener; however, since our sample code does not allocate any resources for the listener, we will leave this method empty (not shown).

Code Sample

The following sample shows an implementation of DOFDomain.StateListener with the system being created in the *stateChanged* method, and the system available callback being used to return the created system to the application developer.

```
private class CustomDomainListener implements DOFDomain.StateListener {
   private final DOFSystem.Config systemConfig;
   private final SystemAvailableCallback systemAvailableCallback;

   public CustomDomainListener(DOFSystem.Config systemConfig,
            SystemAvailableCallback systemAvailableCallback) {
      this.systemConfig = systemConfig;
      this.systemAvailableCallback = systemAvailableCallback;
   }

   @Override
   public void stateChanged(DOFDomain domain, State state) {
      String systemName = systemConfig.getName();

      if(state.isConnected()){
         try{
            DOFSystem system = dof.createSystem(systemConfig, TIMEOUT);
            systemMap.put(systemName, system);
            systemAvailableCallback.systemAvailable(systemName, system, null);
         } catch (DOFException e) {
            System.err.println("Unable to create system. Getting error " +
                     e.getMessage());
            systemAvailableCallback.systemAvailable
                     (systemName, null, e);
         }
      }
   }
```

Code Discussion

Note the following about the sample code shown:

- The constructor for our implementation takes two parameters:
  - **DOFSystem.Config.** The method shown in the next section builds this configuration and passes it to the constructor.
  - **SystemAvailableCallback**. The application developer passes an instance of this callback to the method shown in the next section, and it then passes it to this constructor.
- In the *stateChanged* method, we check the connected state of the domain. If it is connected, we attempt to create the system. If the system is successfully created,

we pass the system to the application developer. If not, we pass an exception to the application developer.

Write Method to Initiate System Creation

In this section, we will examine an example of a method a system developer could create for an application developer to call to initiate system creation.

Code Sample

Our example method builds a system configuration, instantiates DOFDomain, and adds a state listener.

```
public void beginCreateSecureDOFSystem(String name, DOFCredentials
         systemCredentials, SystemAvailableCallback systemAvailableCallback){
   DOFSystem.Config systemConfig;

   if(dof != null){
      systemConfig = new DOFSystem.Config.Builder()
         .setCredentials(systemCredentials)
         .setName(name)
         .build();

      DOFDomain.Config domainConfig = new DOFDomain.Config.Builder
               (systemCredentials)
         .build();

      DOFDomain domain = dof.createDomain(domainConfig);

      domain.addStateListener(new CustomDomainListener(systemConfig,
               systemAvailableCallback));
   } else {
      System.err.println("Unable to create the system. The DOF had not been
               created.");
   }
}
```

Code Discussion

The method requires the application developer to pass a name for the system, the system's credentials, and an instance of the system available callback. The only new Java OAL API in the sample is the DOFDomain.addStateListener method, which activates the DOFDomain.StateListener instance created earlier.

Implement System Available Interface

This section shows how an application developer might implement the system available callback.

Code Sample

```
instanceOfSystemDevClass.SystemAvailableCallback callback = new
         instanceOfSystemDevClass.SystemAvailableCallback() {
   @Override
   public void systemAvailable(String name, DOFSystem system,
            Exception exception) {
      if(system != null){
         requestor = new Requestor(system);
```

```
            showUI();
            return;
         }

         if(exception != null){
            //Fail gracefully.
         }
      }
   };
```

Code Discussion

The application developer must instantiate the system developer’s class and implement the SystemAvailableCallback created earlier.

If the application developer receives a DOFSystem instance, he or she then attempts to create an object that can run DOF operations. Our example assumes the application developer has created a class called Requestor that contains such functionality.

If the system was not created, the application developer should handle the exception. No example is shown because how to handle an exception relies largely on other application functionality.

Call Method to Initiate System Creation

The application developer would initiate all the functionality shown for this use case by calling the method we created to initiate system creation, as shown in the following code:

```
instanceOfSystemDevClass.beginCreateSecureDOFSystem("requestor",
         requestorCredentials, callback);
```

Chapter 5: Permissions

Ultimately, permissions are controlled within the domain storage and not at the connectivity programming level. However, there are optional use cases for associating a DOFConnection or DOFServer instance with a permission set. Before discussing these use cases, it is important to understand a number of permission types.

Permission Types

The OAL defines a number of permission types. This guide will discuss the following:

- Binding Permission
- ActAsAny permission

Binding Permission

The binding permission (DOFPermission.Binding) controls the types of operations a node can perform that involve interface items. Actions are added to this permission to specifically control the operations:

- **Provide.** A provider must be granted this action for the node to provide an interface.
- **Read.** A requestor must be granted this action for the node to be able to perform *get*, *subscribe*, and *register* operations.
- **Write.** A requestor must be granted this action for the node to be able to perform *set* operations.
- **Execute.** A requestor must be granted this action for the node to be able to perform *invoke* operations.
- **Session.** A requestor must be granted this action for the node to be able to open sessions.

Any proxy nodes between the requestor and provider require the permissions of both the provider and requestor for operations to complete successfully.

The permissions matrix more fully illustrates the interaction between permissions and operations on various nodes.

ActAsAny Permission

The ActAsAny permission allows a proxy node to forward operations for other nodes. This permission is instantiated by calling its constructor: DOFPermission.ActAsAny.

Uses Cases for Permission Sets

As stated earlier, the AS ultimately controls the permissions a component can be granted, because the component's credentials are associated with permissions in the domain storage. You cannot use a permission set to request permissions that are not already associated with the credentials in the domain storage; they will not be granted. However, associating credentials with a connection or server is useful for two reasons:

1. **Optimization.** If you do not associate any permissions with the connection or server, then each time an operation that requires a new permission is transmitted, communication occurs with the Authentication Server (AS). However, if the connection or server requests a permission set before sending operations, it cuts down on network traffic associated with granting permissions each time a new operation is initiated.
2. **Greater restriction.** The permission set can be used to further limit the permissions a component has. If you provide a permission set and do not allow the OAL to extend them, the connection or server will have only the specific permissions you specify, regardless of whether the credentials used are associated with additional permissions in the domain storage. This use case is unusual and should probably be avoided when possible to simplify troubleshooting.

Create a Binding Permission

Binding permissions are created using the builder pattern. In addition to allowing you to add any of the actions described above, the DOFPermission.Binding.Builder includes methods for doing the following:

- Limiting the permission to a single object identifier (OID) or to a list of OIDs
- Limiting the permission to a single interface identifier (IID) or to a list of IIDs
- Requiring that OIDs have a specific attribute or a combination of attributes or attribute types before the permission will be granted

If none of these methods are used, the permission set automatically requests permission for all OIDs, IIDs, and attributes that the AS would otherwise grant. Use of these methods is not demonstrated in this guide.

Code Samples

The following sample shows a binding permission for a provider:

```
DOFPermission providerBindingPerm = new DOFPermission.Binding.Builder
          (DOFPermission.Binding.ACTION_PROVIDE)
   .build();
```

The following example shows a binding permission that grants only read and write actions for a requestor:

```
DOFPermission requestorBindingPerm = new DOFPermission.Binding.Builder
          (DOFPermission.Binding.ACTION_READ)
   .addActions(DOFPermission.Binding.ACTION_WRITE)
   .build();
```

The following sample shows the permissions a proxy would need to forward operations between nodes with the permissions in the previous samples:

```
DOFPermission proxyBindingPerm = new DOFPermission.Binding.Builder
         (DOFPermission.Binding.ACTION_READ)
   .addActions(DOFPermission.Binding.ACTION_WRITE)
   .addActions(DOFPermission.Binding.ACTION_PROVIDE)
   .build();
```

Note: In a binding permission, if the AS denies any of the actions in the permission, the entire permission is denied. For example, if the above example of a requestor was granted the *read* action by the AS, but denied the *write* action, the entire permission would be denied.

Create a Permission Set

Permission sets are also created using the builder pattern. The samples in this section show how to place the binding permissions created in the previous section into permission sets. In addition, the proxy node's permission set also contains the ActAsAny permission.

Code Samples

The following is a permission set for a provider:

```
DOFPermissionSet providerPermissions = new DOFPermissionSet.Builder()
   .addPermission(providerBindingPerm)
   .build();
```

The following is a permission set for a requestor:

```
DOFPermissionSet requestorPermissions = new DOFPermissionSet.Builder()
   .addPermission(requestorBindingPerm)
   .build();
```

The following is a permission set for a proxy:

```
DOFPermissionSet proxyPermissions = new DOFPermissionSet.Builder()
   .addPermission(proxyBindingPerm)
   .addPermission(new DOFPermission.ActAsAny())
   .build();
```

Build a Configuration

After building a permission set, it can be added to either a connection or server configuration.

Code Sample

The following sample shows how to add a permission set to a connection configuration.

```
DOFConnection.Config myConnConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, myAddress)
   .setCredentials(providerCredentials)
   .setPermissions(providerPermissions)
```

```
    .setPermissionsExtendAllowed(false)
    .setSecurityDesire(SecurityDesire.SECURE)
    .build();
```

Code Discussion

The *setPermissions* method of the builder takes a permission set as a parameter and is used to add a permission set to a configuration. The builder for a server configuration has an identical method.

The *setPermissionsExtendAllowed* method is used to limit the component's permissions to only those specified in the set. So in this example, the connection would have only the provide action in a binding permission, even if the AS grants other permissions to the credentials. The default setting for this parameter is true.

Access Denied Exception

At this point, if you attempt to perform operations, you may begin to see the "access denied" exception. This exception means that the node originating the operation (usually a requestor) is not being granted the permission it needs to complete the operation. (If a proxy or receiving node does not have the correct permission, the originating node receives a TIMEOUT exception.)

As a developer, you should check for connection or server configurations where *setExtendAllowed* is set to false and ensure that the proper permissions and actions are granted in the permission set. If you have no such configurations, the problem is likely to be in the domain storage, and you can escalate the problem to the domain manager.

Chapter 6: Bridging Security Domains

If you need components to communicate between two different domains, you can create a bridge and add it to a connection or server. The AS created for training module 4 provides two domains. To experiment with code in this chapter using the training AS, you must run the module 4 AS and create additional credentials using the following domain and authentication identifiers (see *Creating Credentials*):

- **Domain.** [6:2.tech-services.opendof.org]
- **Provider.** [3:provider@2.tech-services.opendof.org]
- **Requestor.** [3:requestor@2.tech-services.opendof.org]
- **Proxy.** [3:proxy@2.tech-services.opendof.org]

The credentials for the provider, requestor, and proxy all use the same key (all zeros) that was used for credentials created earlier. In this chapter, we will refer to the domain we have been using in previous chapters as domain 1. We will refer to the new domain as domain 2. Sample code shown in this chapter uses only the provider credentials and providerCredentials2 as the variable name for its credentials in domain 2.

Bridge Behavior

Bridges can be added to connections or servers; however, the behavior on each is slightly different:

- **Servers.** A server can have credentials in multiple domains, and its credentials determine which domains it can accept connections from. On a server, a bridge ignores incoming secure connections. The bridge routes incoming unsecured connections to the domain that the server's bridge has credentials for. A server bridge is only capable of bridging unsecured traffic to a secure domain. A server must have its security desire set to NOT_SECURE or ANY to create this type of bridge. A server with a SECURE security desire ignores incoming unsecured connections and the bridge will not function.
- **Connections.** For communication with external nodes, a connection always connects to the domain it has credentials for. If given a bridge, the connection is in the bridge's domain within its originating node. A connection with credentials in one domain and a bridge that has credentials in a different domain creates a bridge between the two domains. A connection bridge can be used either to bridge traffic between two secure domains or to bridge traffic from the unsecured domain to a secure domain.

This chapter will focus on connection bridges, because they are more flexible and can bridge secure domains. The process for adding a bridge to a server is similar and can be extrapolated from the sample code shown for a connection.

Creating a Bridged Connection

Adding a bridge to a connection requires using the builder pattern to instantiate a bridge and then setting the bridge in the connection configuration. We will show examples of how to bridge secure domains and how to bridge the unsecured domain to a secure domain.

Bridging Secure Domains

The sample code in this section shows how to create a bridge between domain 1 and domain 2.

Code Sample

```
DOFOperation.Bridge.Config bridgeConfig = new
         DOFOperation.Bridge.Config.Builder()
   .setCredentials(providerCredentials)
   .build();

DOFConnection.Config myConnectionConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, myAddress)
   .setCredentials(providerCredentials2)
   .setBridge(bridgeConfig)
   .setSecurityDesire(SecurityDesire.SECURE)
   .setTrustedDomains(DOFObjectID.DOMAIN_BROADCAST)
   .build();
```

Code Discussion

Note that the bridge in the previous sample uses the provider credentials from domain 1. A DOFSystem on the same node would need credentials in domain 1 to use this connection.

The credentials set in the connection configuration are the provider credentials from domain 2. This connection would route traffic from a DOFSystem in domain 1 to external nodes in domain 2.

If the DOF is set to be a router (for example, on a proxy), the connection can also route traffic from other connections or servers in domain 1 to domain 2.

Bridging the Unsecured Domain to a Secure Domain

The sample code in this section shows how to bridge unsecured components to a secure domain. On endpoint nodes in a network with only a single DOFSystem instance and very few DOFConnection instances, this can be an alternative to creating secure systems. However, this alternative must be used carefully to ensure that no holes are created in security.

Code Sample

```
DOFOperation.Bridge.Config bridgeConfig = new
         DOFOperation.Bridge.Config.Builder()
   .setCredentials(null)
   .build();
```

```
DOFConnection.Config myConnectionConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, myAddress)
   .setCredentials(providerCredentials2)
   .setBridge(bridgeConfig)
   .setSecurityDesire(SecurityDesire.SECURE)
   .addTrustedDomains(DOFObjectID.DOMAIN_BROADCAST)
   .build();
```

Code Discussion

Setting the credentials in the bridge to null allows the connection to accept traffic from unsecured systems on the DOF. Because the connection has credentials in domain 2, traffic is routed to other nodes in domain 2.

Chapter 7: Conclusion

After reading this guide, you should have the basic knowledge required to connect to an Authentication Server (AS), create secure DOF components, add permission sets to connections and servers, and create bridges between security domains.