
Securing an EJB Application in BES

5.2

Krishnan Subramanian

Exploring BES security in Borland Enterprise Server 5.2

1. Introduction

Security is admittedly a vast and often complex topic to deal with. This little writeup takes a no-nonsense approach to setting up a basic security model with the EJB Container. This paper will not explain any of the theoretical concepts behind JAAS - and assumes that the reader is familiar with these technologies.

Some of the documents and papers that are required reading prior to getting your hands dirty on the contents of this article are:

- BES 5.2.0 Developer's Guide and User's Guide
- JAAS documentation [<http://java.sun.com/products/jaas/>]
- BES security examples ([\\$BES/examples/security](#))
- BES Security FAQs at:
 - SecurityFAQ [<http://info.borland.com/devsupport/bes/faq/5.2/security/SecurityFAQw52.html>]
 - Security Properties [<http://info.borland.com/devsupport/bes/faq/5.1/security/props.html>]

We are going to take the simplistic case of a standalone Java client communicating with an EJB Container. The next installment of this paper (part 2) will show you how to extend this model - where the client is a web application running within BES's Web Container.

Over-enthusiastic developers might end up (mis)configuring things so that the server's partition services do not even start up anymore. I'd highly recommend taking backups of any property file prior to modifying it.

1.1. Application Model

While it is possible to take an existing application and secure it with little or no changes to code, it is imperative that security mechanisms and techniques be a part of the design and development process rather than be implemented as an afterthought or during the last stages of development.

The model is:

```
                                RMI/IIOP
Java Standalone App <-----> EJB Container
```

The design philosophy is to employ a security mechanism whereby clients are required to first authenticate themselves to the server (say via a username/password combination). Once authenticated, clients then receive an identity - and invocation of business methods on the server should be allowed iff that client is authorized to call that method.

Let's take the simple case of a stateless session bean (SLSB) invoked by a client. The SLSB is a Calculator; with (business) methods to add/subtract integers. Let's assume that the Calculator SLSB has two business methods:

- `public int add(int a, int b)`
- `public int subtract(int a, int b);`

A simple Java standalone client can then be developed to invoke these methods. In a unsecured scenario; assuming that the SLSB has remote interfaces, any client can invoke these methods.

The idea is to secure the application by simple means (at least in the context of this example and paper) so that

clients are required to log on first. From then on, depending on method permissions, clients may or maynot be authorized to invoke particular methods. Let's take the case where clients in the role of a 'accountant' are allowed to invoke both add() and subtract() methods while clients in the role of a 'clerk' are only allowed to invoke the 'add' method.

We need a backend capable of maintaining a list of usernames, passwords and associated role(s). These would then be consulted when the client logs on and/or invokes methods to ensure that access is indeed authorized. We'll use a database for this purpose; the interface to which is provided by the BasicLoginModule. (See BES Developer's Guide for information on other LoginModules and how/when they could be used).

So, our model now is:

```
Java Standalone App <-----> EJB Container <-----> DB (for storing user and role info)
```

1.2. Creating the User database for realm

As a first step, we'll create/configure our database to store users and roles. Borland provides the 'userdbadmin' tool (run from the command line) to auto-create required tables, create groups and associate users with groups.

We'll use JDataStore for this example; though any backend - like Oracle, DB2, Sybase, MS SQL Server, etc. can be used. A sample command is shown below. For JDataStore, the command is to be run from the command prompt when the current working directory is:

```
$BES/var/servers/<server_name>
userdbadmin
-create
-db jdbc:borland:dslocal:adm/security/mydb.jds
-driver com.borland.datastore.jdbc.DataStoreDriver
-user admin
-password admin
-interactive

>addgroups accountant
>addgroups clerk
>adduser krish krishpwd accountant
>adduser john johnpwd accountant
>adduser bill billpwd clerk
>adduser scott scottpwd clerk
>quit
```

The above commands typed at the ">" prompt creates two groups 'accountant' and 'clerk' in the database. Two users with usernames krish and john are in an accountant role; while bill and scott are in the role of clerks. (Type 'help' at the ">" prompt for a list on available commands).

1.3. Configuring Sever Security

1.3.1. Enabling Security

The first step is enabling security on the Server. This can be done by starting up BES; followed by the console, and then from the console, right-click the partition, choosing 'Configure' and then selecting the 'Security' tab. Within this dialog:

- enable security
- enable SSL (for secure communication between clients and servers)

1.3.2. Configuring Authentication

The second step to let the server-side know that the above database is to be associated with a particular Realm (see JAAS docs for more details). Click on the 'Authentication' button on the 'Security' tab. The config.jaas for that partition is then displayed. Add your realm entry to this along the lines of:

```
MyRealm {
  com.borland.security.provider.authn.BasicLoginModule required
```

```

DRIVER=com.borland.datastore.jdbc.DataStoreDriver
URL="jdbc:borland:dslocal:../../adm/security/mydb.jds"
TYPE=BASIC
LOGINUSERID=admin
LOGINPASSWORD=admin;
};

```

The above settings are consulted when the client logs on. The username/password by the client are verified against the above entry when a client identity is to be assigned.

1.3.3. Configuring Authorization

The third step is to configure Authorization. Here, under the security tab, click the 'Authorization' button. While it is possible to add rolemap entries to the existing default.rolemap file, we're going to create a new authorization domain.

Right-click the 'Security Domains' element in the tree view and create a new domain called 'MySecureDomain' (or whatever). The next steps are to now create roles within this rolemap file and associate those with groups and/or users in the database.

We'll create two roles in the 'MySecureDomain' called 'accountant_role' and 'clerk_role' and these would then map to the 'accountant' and 'clerk' groups in the database. The MySecureDomain.rolemap would be along the lines of:

```

accountant_role {
  *GROUP=accountant
}
clerk_role {
  *GROUP=clerk
}

```

The rolemap editor dialog provides easy configuration and customization of the rolemap file by providing wizards for creating roles and entering rules and attributes.

The 'Security' tab configuration is now complete, but we still need to set a few VisiBroker properties to complete security related settings.

1.3.4. Configuring Other security properties

For this, select the VisiBroker tab, and hit the 'Advanced' button. This opens up the vbroker.properties file for that partition. Find the line in this file that reads:

```
vbroker.security.authDomains=default
```

The domain we added (MySecureDomain) has to be listed in the above property. So the above statement should be modified to read:

```
vbroker.security.authDomains=default,MySecureDomain
```

Next, we need to associate the MySecureDomain with the rolemap file we just created. This can be done by first finding the lines in the partition's vbroker.properties file that read:

```

# How to handle requests for methods not in the rolemap file - (grant|deny)
vbroker.security.domain.default.defaultAccessRule=grant

# (this path is relative to server.instance.root/adm/properties/partitions/partition.name)
vbroker.security.domain.default.rolemap_path=security/default.rolemap

```

Now, add the following properties for the MySecureDomain below the existing properties mentioned above:

```

vbroker.security.domain.MySecureDomain.defaultAccessRule=grant
vbroker.security.domain.MySecureDomain.rolemap_path=security/MySecureDomain.rolemap

```

See docs for above properties and additional ones pertaining to configuration of authorization domains.

That concludes BES Server configuration. Restart of partition(s) might be necessary for settings to take effect. Ensure that the EJB Container starts up correctly after a restart. Look at the partition log files to possibly help lo-

cate incorrect/invalid configuration settings.

1.3.5. Securing your EJBs

Next, we configure the EJB module to take advantage of the domain we just added. Open up the EJB module using the deployment descriptor editor (DDE). Go to the toplevel structural information for that archive, and then in the textfield for the 'Authorization domain', enter 'MySecureDomain' (without the quotes). Open up the 'Security Roles' element in the DDE and add two roles called:

```
accountant_role
clerk_role
```

These should correspond to entries in the MySecureDomain.rolemap file we created earlier (that is, the MySecureDomain authorization domain).

The above roles once defined are then available to method permission entries. To define method permissions, drill down to the 'Method Permissions' of the Calculator SLSB in the DDE. Add four entries. Ensure that the entries look like:

Table 1. Permissions

Interface	Method	Access	accountant_role	clerk_role
Home	*	Unchecked		
Remote	add(int, int)	By role	yes	yes
Remote	subtract(int, int)	By role	yes	yes
Remote	remove()	Unchecked		

That concludes securing the EJB application. Deploy this archive to the partition.

1.3.6. Configuring security for the Client

Now try running the test client. The test client should receive a CORBA INV_OBJREF 0[] exception.

This is because the EJB application is now secure and unsecure clients cannot communicate with the server application.

The client application code does not need to change, but certain properties need to be passed to the client (as JVM params). These can be stored in property files as well. First create a file called client.props and store the following VisiBroker properties in that file:

```
vbroker.security.disable=false
vbroker.security.login=true
vbroker.security.login.realms=MyRealm
vbroker.security.authentication.callbackHandler=com.borland.security.provider.authn.HostCallbackHa
```

This tells the client to acquire authentication information for the realm MyRealm. Now the client application can be started via:

```
% vbj -DORBpropStorage=./client.props MyTestClient
```

The client application then gets prompted to enter the username/password combination for 'MyRealm'. Entering a valid combination such as: krish/krishpwd or scott/scottpwd etc, will result in the client invoking methods on the server if authorized to do so. An unauthorized access will result in a org.omg.CORBA.NO_PERMISSION wrapped within an java.rmi.AccessException.

1.4. Summary

Well, that's about it. You could create fancy CallbackHandlers to prompt users to enter information say from within a dialog/frame. Or you could use different LoginModules to authenticate/authorize against a different backend (such as a NT/Unix domain, LDAP, etc.).

2. Part II

In Part I of this document series, we secured Borland Enterprise Server and had the EJB Container use those security mechanisms to authenticate and authorize access for standalone Java clients.

In this article (Part II), we run through the steps involved in [re]using the same security mechanism to authenticate and authorize web-based access.

In Part I, we created a database (JDataStore as an example; though any RDBMS such as Oracle, Sybase, MS SQL Server etc, could have been used) that contained a list of groups and users associated with those groups. We then configured Borland Enterprise Server to point to this database (via a Realm, Rolemap file and certain VisiBroker properties).

We will now reuse the exact same configuration to first authenticate web clients. That is, clients when accessing a protected URL through their web browser are prompted to enter a username and password for that realm. The username and password entered via the web browser are checked against those present in the role database configured earlier.

If the username and password for that realm are found; then access to that protected URL is allowed. For now, to keep things simple we do not deal with authorization at this point in time.

Again, we will use Container managed security for the Web Container (Tomcat) in Borland Enterprise Server. It is possible for Tomcat to connect to an existing database containing roles and users and use that configuration for authenticating users. Tomcat (by default) provides a few plug-ins that can be used to provide access to certain backends for authentication. However, it should be noted that the Servlet (and for that matter - the EJB) specifications while providing a portable mechanism for applications to declare their security requirements (through associated deployment descriptors) do not define any of the interfaces or APIs associated with the communication/implementation between a [Web/EJB] Container and the authentication backend.

Borland has extended Tomcat to allow it to use the same backend defined in Borland Enterprise Server - the implementations of which are provided by various LoginModules (as defined in the JAAS specification). So, only a Tomcat bundled with Borland Enterprise Server - Web edition (included with the VisiBroker and AppServer editions) will be able to communicate with and use the security implementations provided by BES.

2.1. Securing Tomcat

Once the Borland Enterprise Server has been set up to authenticate against a backend (as described in Part I), we will now secure the Web Container. The first step is to edit the Web Container's server.xml file and change the realm that Tomcat uses (by default, the MemoryRealm) to use Borland Security Service Realm (BSSRealm).

For this, from the BES Console, right-click the Web Container, choose 'Configure' and run down to the 'Realms' tab. For the HTTP Service, enter the following class name for the BSSRealm:

```
org.apache.catalina.realm.BSSRealm
```

The secure realm we set up in Part I was 'MyRealm', and so enter this for the 'Login Domain'. These entries will reflect in Tomcat's server.xml file under the HTTP Service tag:

```
<Realm name="MyRealm" className="org.apache.catalina.realm.BSSRealm" />
```

2.2. Securing the Web Application

Our objective is to authenticate web clients who request a protected URL (defined through deployment descriptors as we shall see) against the backend setup for authentication and authorization in Borland Enterprise Server.

This can be done by opening up the deployment descriptor editor (DDE) for that Web Application's WAR file. Go to the toplevel structural information for that archive, and then in the textfield for the 'Authorization domain', enter 'MySecureDomain' (without the quotes).

Open up the 'Security Roles' element in the DDE and add two roles called:

```
accountant_role  
clerk_role
```

These should correspond to entries in the MySecureDomain.rolemap file we created earlier (that is, the MySecureDomain authorization domain) in Part I of this document series.

Next, open up the 'Security Constraints' element in the DDE and add a security constraint called (say) 'MyWebAp-

pConstraints'. As the authorized roles for this domain, check both 'accountant_role' and 'clerk_role'.

We now need to define which URLs are protected. To 'MyWebAppConstraints' in the DDE, right-click and add a new Web Resource Collection and give it a name such as 'MySecureColl'. For this Collection, add a URL pattern:

```
/*
```

For the above URL pattern, associate it with both the HTTP GET and POST methods.

We now need to tell the web application the name of the realm used for authentication. In the DDE, when the toplevel structural information for that archive is selected, open up the 'Login' tab on the right. Select the 'Basic' mechanism and in Realm name enter 'MyRealm'. Again, the name of the realm corresponds to that used in Part I of the document series when setting up security for Borland Enterprise Server.

Deploy the web application to Borland Enterprise Server's partition. Once this is done, point your web browser to the web application and try accessing one of the protected URLs (<http://hostname:8080/MyWebApp/MyFile.html>).

The browser will prompt you to enter a user name and password via a dialog prompt. Entering a valid combination (krish/krishpwd, scott/scottpwd) will result in access to the requested web resource. That concludes basic authentication mechanisms. Other mechanisms to authenticate web clients can be employed - such as a login page which is form based or using certificates etc.,

We will now show how security identities can propagate from the Web Container to the EJB Container seamlessly. In Part I of the document series, we showed how standalone Java clients accessed a secure EJB Container. Access in that case was based on roles that clients were assigned and how method permissions on EJBs were defined in a declarative manner. The same principles apply to web-based clients as well. Once a successful authentication has occurred, the Web application (servlets, JSPs etc,) can invoke methods on a secure EJB Container. Depending on the role that the web client is in, the EJB Container will allow or deny access.

Create a servlet/jsp as a part of the web application that invokes methods on the CalculatorSession EJB (Part I of this document series). When the client accesses the servlet/jsp via the web browser, the client is first authenticated against Borland Enterprise Server's security realm. This authentication takes place when the user enters his username and password for that realm from the browser.

When the servlet/jsp invokes methods on the EJB, the identity is propagated to the EJB Container where authorization occurs depending on the role(s) of the web client and EJB method permissions defined.

If the Web and EJB containers are not colocated in the same partition, then the call will fail. In fact this is true for any (secure) call that crosses partitions - for example (a) when an Servlet/JSP in one partition invokes a method on an EJB in another partition or (b) when an EJB in one partition invokes a method on an EJB in another partition.

In fact, in the above scenario, the caller identity is always propagated irrespective of whether the J2EE/CORBA application is colocated or not. The reason the calls fails with a security exception in the non-colocated scenario is because the called partition does not trust the caller partition.

So, the basic mechanism to solve the issue is for the called partition to authenticate the caller partition and not authenticate the credentials of the user again - since the user has already been authenticated by the first tier (caller partition). In effect, only the first tier that the end user accesses needs the (user) authentication entries - the Realm and LoginModule configuration. End tiers do not authenticate the user, but only the caller partition and apply authorization checks on the user 's identity based on roles defined in the authorization domain's rolemap file.

To configure the end tiers to trust a caller partition, the end tiers need the following configuration information: In the application's Authorization Domain rolemap of the end tier partition, enter a role:

```
Trusted_Partition {
    *(CN=admin, REALM=ServerRealm)
}
```

And in the vbroker.properties file of the end tier partition, enter the following property

```
vbroker.security.assertions.trust.1=Trusted_Partition@MyDomain
vbroker.security.assertions.trust.2=Another_Partition@AnotherDomain
vbroker.security.assertions.trust.3=<.....>
vbroker.security.assertions.trust.4=<.....> and so on
```

Now, if the application is not colocated, the end tier will receive the user identity (which has been authenticated

by the first tier) and the called partition only performs the authorization checks on whether the resource is callable by the role(s) that user is in.

That's about it.

Some interesting links are:

- [\\$BES/examples/security/customlogin](#) (for how to write and plugin your own login modules)
- [\\$BES/examples/security/securecart](#)
- [\\$BES/examples/security/trustpoints](#) (for how to use certificates and trust points)
- Writing your own callback handler is illustrated in some of JAAS guides/examples from Sun. <http://java.sun.com/products/jaas>