Role-Based Security

Doug Hennig

Role-based security allows you to specify which users have rights to particular secured entities, but at a group, or role, level. This month, Doug Hennig starts a discussion of how he implements role-based security in his applications.

For my 100th article (yes, it has been that long!), I'm going to start a multi-part series on something I recently added to one of my applications: role-based security.

In the September through November 2004 issues of FoxTalk, Andy Kramek and Marcia Akins discussed this topic in great detail. So why cover it again? Although their design was great and I learned a lot from it, I need a different approach. In particular:

- I need users to be in more than one role. The administrator of the application isn't necessarily a power-user; they're just the person who has access to the form where rights are assigned to different roles. Although it's unlikely, it's possible that a department secretary could be the application administrator. That doesn't mean she has the right to see salary data in a payroll table, however. So, the secretary may also be in an "administrative staff" role, which allows access to parts of the application and restricts access to other parts.
- Andy and Marcia use security by exception, in which a user is assumed to have full access to something unless otherwise specified. For a variety of reasons, I need the opposite; a user has no access to something unless they are specifically assigned rights to it.
- Their design implements field-level security, which I also need, but in a different way. Rather than securing the controls bound to fields in forms by hiding or disabling them, my application needs to know what fields the user can see in reports (if a user can't see any of the fields in a report, they can't see the report at all).
- Finally, I want to expose all methods that maintain security, such as adding users, changing a user's role, and so forth, as methods of a security manager class so these tasks can be done programmatically, such as through a COM object, if necessary.

Although I'd rather use existing code than reinventing the wheel, my needs were too different from Andy and Marcia's design, so I had to start from scratch (well, not quite—as I mentioned earlier, I did get some great ideas from their design).

I'd like to refresh your memory about what a role is. Andy and Marcia had a very clear definition: a role is an identifier for a specific access pattern that could be used by one or more users of an application. It isn't necessarily related to a specific job description or user. So, with that in mind, let's look at my implementation.

Data model

As you can see in **Figure 1**, there are five tables involved in my implementation of role-based security:

- USERS: contains information about each user in the system, such as user name and password (which is encrypted, of course).
- ROLES: contains information about the roles defined in the system.
- USERROLES: provides a join table between the users and roles tables to resolve their many-to-many relationship.
- ELEMENTS: contains information about secured objects. I called secured objects "elements" because there could be lots of different secured objects: forms, fields, reports, menu functions, and so forth. This table could be used in an administrator dialog where rights to certain elements are assigned to particular roles.

- SECURITY: defines what rights each role has to a particular element.

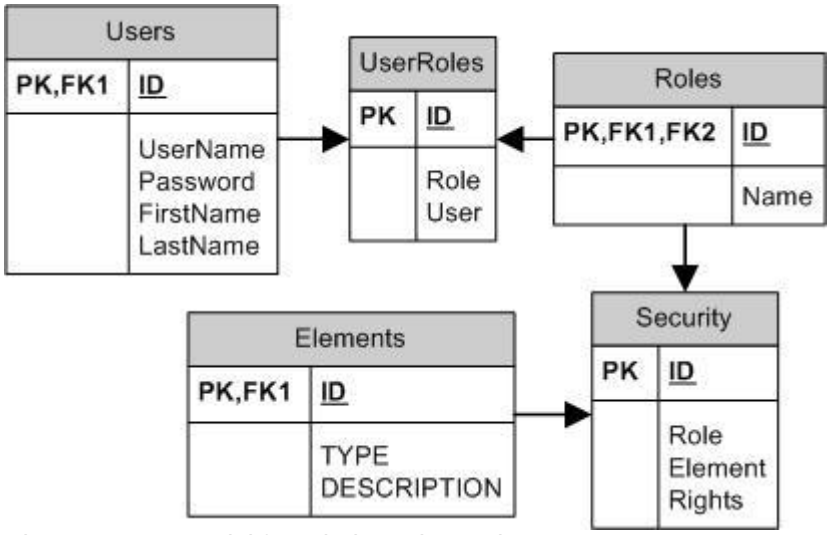

Figure 1. Data model for role-based security.

User and role collections

Although we could access users and roles through their records in the tables, it makes sense to expose them as members of collections instead, especially if we want to use this in a COM object. The SFUserCollection class is a collection of user objects, each of which contains properties about a specific user. Similarly, SFRoleCollection is a collection of role objects, which contain properties about the roles. Both of these classes are contained in SFSecurity.VCX.

Before we look at these classes, let's examine their common base class, SFCollectionTable. SFCollectionTable, defined in SFCtrls.VCX, is a subclass of SFCollectionOneClass, which is a subclass of SFCollection, which is a subclass of Collection. SFCollectionTable is specially designed to act as an interface for records in a table. The members of the collection are objects, one per record in the table, with properties that map to the columns in the table.

The FillCollection method is used to load the collection from the table it's associated with. FillCollection calls OpenTable (which we won't look at) to open the table specified in the cTable property and put its alias into cAlias. FillCollection then spins through the table, calling the CreateEntityObject method for each record to create an object with properties mapping to the fields in the record. CreateEntityObject is abstract in this class, but could be as simple as RETURN SCATTER loObject or as complicated as instantiating an object and setting its properties to the values of the fields in the current record. FillCollection then calls GetEntityName, which is also abstract in this class, to get the name we want to use when we add the object to the collection, and then finally calls Add to store the object.

```
local lnSelect, ;
  loEntity, ;
  lcName, ;
  llReturn
if This.OpenTable()
  lnSelect = select()
  select (This.cAlias)
  scan
    loEntity = This.CreateEntityObject()
    lcName   = This.GetEntityName(loEntity)
    This.Add(loEntity, lcName)
  endscan
  select (lnSelect)
  llReturn = .T.
endif This.OpenTable()
return llReturn
```

The Add method is overridden in SFCollectionTable. Add can be called a couple of ways: by passing an object to be stored and the name to use as its key, which is how it's called from FillCollection, and by passing just the name, which is how it's typically called from client code. In the latter case, we don't just add an object to the collection, but also add a new record to the table.

```
lparameters tuItem, ;
  tcKey
local loEntity
do case

* If we were passed a name, add a record to the table
* and use the default behavior to add it to the
* collection. Note that we re-read loEntity from the
* collection in case we were specified a duplicate
* name and it isn't added to the collection.

  case vartype(tuItem) = 'C'
    loEntity = This.AddRecord(tuItem)
    loEntity = dodefault(loEntity, tuItem)

* We were passed an object, so add it to the
* collection. As above, we re-read the object from
* the collection in case it's a duplicate.

  case vartype(tuItem) = 'O'
    loEntity = tuItem
    loEntity = dodefault(loEntity, tcKey)

* Invalid parameters.

  otherwise
    loEntity = .NULL.
    throw 'Function argument value, type, or ' + ;
      'count is invalid.'
 endcase
nodefault
return loEntity
```

Other methods are also overridden or added in SFCollectionTable, including Remove and RemoveRecord, but that's all we'll look at.

SFUserCollection is a subclass of SFCollectionTable. Its cTable property is set to USER.DBF by default, but of course it could be changed to point to a different users table.

Since I didn't need the user object to have any behavior, the CreateEntityRecord method in this class uses SCATTER NAME to create an object with properties matching the fields. Notice I'm not decrypting the password, which is encrypted in the table, here. The problem with doing that is the password is then available in clear text in memory, which could make it possible for a hacker to find out the passwords. We'll discuss password encryption next month.

```
local lnSelect, ;
  loUser
lnSelect = select()
select (This.cAlias)
scatter name loUser
with loUser
  .UserName  = trim(.UserName)
  .Password  = trim(.Password)
  .FirstName = trim(.FirstName)
  .LastName  = trim(.LastName)
endwith
select (lnSelect)
return loUser
```

The AddRecord method, called from the parent class' Add method to add a new record to the table, checks whether the specified name already exists, and if not, adds it. It then calls CreateEntityObject to create a user object from the record.

```
lparameters tcName
local loUser
if not seek(upper(padr(tcName, ;
  len(__USERS.USERNAME))), '__USERS', 'USERNAME')
  insert into __USERS (USERNAME) values (tcName)
endif not seek(upper(padr(tcName ...
loUser = This.CreateEntityObject()
return loUser
```

SFUserCollection has a new method, GetUserByID. Since the key for a user object in the collection is the user name, finding a user by ID is more difficult. To make it easier, GetUserByID simply SEEKs the ID in the table and if it's found, calls CreateEntityObject to create the user object for the user record.

```
lparameters tiUserID
local loUser
if seek(tiUserID, '__USERS', 'ID')
  loUser = This.CreateEntityObject()
else
  loUser = .NULL.
endif seek(tiUserID, '__USERS', 'ID')
return loUser
```

We've seen how new records are added to the users table, but how are changes made to properties of the user objects saved, such as when a user's password is changed? SFCollection has a SaveCollection method that's called when the collection is destroyed; it can, of course, also be called manually to save the collection on demand. It spins through the collection and calls SaveItem for each member. SaveItem is abstract in SFCollection, but SFUserCollection implements it using GATHER NAME to update the appropriate record in the users table with the values of the properties of the current object.

We won't look at SFRoleCollection because it's very similar to SFUserCollection. Like SFUserCollection, it has a GetRoleByID method to return a role object for the specified ID.

Security manager

It's time to look at the security manager class. We don't have space to cover the entire class in the article; I'll continue the discussion next month.

SFSecurity, in SFSecurity.VCX, is a subclass of SFCustom, my Custom base class defined in SFCtrls.VCX. Its Init method instantiates user and role collection objects into the oUsers and oRoles properties. Notice that rather than directly specifying SFUserCollection and SFRoleCollection, this code gets the name of the classes and their libraries from properties. This is a more flexible design, since I can subclass SFSecurity and specify different classes if I need that. Init also opens VFPEncryption.FLL, an encryption library generously provided to the VFP community by new MVP Craig Boyd, and available for download from his Web site (http://www.sweetpotatosoftware.com/SPSBlog). I'll discuss VFPEncryption.FLL in more detail next month.

```
with This

* Instantiate user and role collection objects.

  .oUsers = newobject(.cUserCollectionClass, ;
    .cUserCollectionLibrary)
  .oRoles = newobject(.cRoleCollectionClass, ;
    .cRoleCollectionLibrary)

* Open the VFP encryption library.

  set library to VFPEncryption.FLL additive
endwith
```

The Setup method opens the security-related tables, instructs the user and role collections to fill themselves with objects from the appropriate tables, and sets the lSetup property to indicate that set up has been performed. It also instantiates a localizer object. I won't go into details on this object in this article; I'll save that for a future article. However, this object is used for localization purposes; its main method,

GetLocalizedString, looks up a string in a resource table and returns the equivalent of that string in the specified language. This allows you to avoid using language-specific strings in your code so you can easily localize it for other languages. Note that Setup can be called manually after instantiating the object, but it's also called automatically from many other methods.

```
with This
  llReturn = .OpenTables()
  llReturn = llReturn and .oUsers.FillCollection()
  llReturn = llReturn and .oRoles.FillCollection()
  if llReturn and vartype(.oLocalizer) <> 'O'
    .oLocalizer = newobject('SFLocalize', ;
      'SFLocalize.VCX')
  endif llReturn ...
  .lSetup = llReturn
endwith
return llReturn
```

The IsUserInRole method returns .T. if the specified user is in the specified role. Notice that both the user and role are specified by ID rather than name; that will be true for most methods in this class. Also notice that if the user ID isn't specified, the ID of the currently logged-in user is used if there is one. We'll look at how a user logs in next month, but for now, know that once a user has logged in, a reference to that user's user object is stored in the oCurrentUser property.

```
lparameters tiUser, ;
  tiRole
local liUser, ;
  llReturn, ;
  lcMessage
with This

* If the user ID wasn't specified, use the one for
* the logged-in user.

  if (vartype(tiUser) <> 'N' or tiUser = 0) and ;
    vartype(.oCurrentUser) = 'O'
    liUser = .oCurrentUser.ID
  else
    liUser = tiUser
  endif (vartype(tiUser) <> 'N' ...
  do case

* Ensure we've been set up properly.

    case not .lSetup and not .Setup()
      llReturn  = .F.
      lcMessage = .GetLocalizedString(.cErrorMessage)

* Ensure the parameters are valid.

    case not .CheckUser(liUser)
      llReturn  = .F.
      lcMessage = ;
        .GetLocalizedString('ERR_INVALID_USER_ID')
    case not .CheckRole(tiRole)
      llReturn  = .F.
      lcMessage = ;
        .GetLocalizedString('ERR_INVALID_ROLE_ID')

* See if the specified user is in the specified role.

    otherwise
      llReturn = seek(str(tiRole) + str(liUser), ;
        '__USERROLES', 'ROLEUSER')
  endcase
endwith
```

```
* Raise an error if we had a problem (do this outside
* the WITH structure to avoid problems with unclosed
* WITHs).

if not empty(lcMessage)
  throw lcMessage
endif not empty(lcMessage)
return llReturn
```

AddUserToRole and RemoveUserFromRole are very simple: they call IsUserInRole to determine if the specified user is in the specified role, and update the USERROLES table accordingly. Here's the code for AddUserToRole.

```
lparameters tiUser, ;
  tiRole
if not This.IsUserInRole(tiUser, tiRole)
  insert into __USERROLES (ROLE, USER) ;
    values (tiRole, tiUser)
endif not This.IsUserInRole(tiUser, tiRole)
```

GetRolesForUser returns a collection of role objects the specified user belongs to. Note that you can indicate whether the ID of the role is used as the key for the items in the collection (normally the name of the role is used) by passing .T. for the second parameter.

```
lparameters tiUser, ;
  tlIDAsKey
local loRoles, ;
  liUser, ;
  lcMessage, ;
  lnSelect, ;
  loRole
with This

* Create a collection for the roles.

  loRoles = newobject('SFCollection', 'SFCtrls.VCX')

* If the user ID wasn't specified, use the one for
* the logged-in user.

  if (vartype(tiUser) <> 'N' or tiUser = 0) and ;
    vartype(.oCurrentUser) = 'O'
    liUser = .oCurrentUser.ID
  else
    liUser = tiUser
  endif (vartype(tiUser) <> 'N' ...
  do case

* Ensure we've been set up properly.

    case not .lSetup and not .Setup()
      lcMessage = .GetLocalizedString(.cErrorMessage)

* Ensure the parameters are valid.

    case not .CheckUser(liUser)
      lcMessage = ;
        .GetLocalizedString('ERR_INVALID_USER_ID')

* Get all records from the user roles table that have
* the specified user, get the appropriate role
* objects from the roles collection, and add them to
* the collection we'll return.

    otherwise
      lnSelect = select()
      select __USERROLES
      scan for USER = liUser
```

```
        loRole = .oRoles.Item(ROLE)
        do case
          case vartype(loRole) <> 'O'
          case tlIDAsKey
            loRoles.Add(loRole, transform(ROLE))
          otherwise
            loRoles.Add(loRole, loRole.Name)
        endcase
      endscan for USER = liUser
      select (lnSelect)
  endcase
endwith

* Raise an error if we had a problem (do this outside the WITH structure to
* avoid problems with unclosed WITHs).

if not empty(lcMessage)
  throw lcMessage
endif not empty(lcMessage)
return loRoles
```

Check it out

Let's check out what we've covered so far. The following code, taken from TestSecurity.PRG, creates a couple of roles, a couple of users, and adds those users to the appropriate roles.

```
loSecurity = newobject('SFSecurity', 'SFSecurity.vcx')
if not loSecurity.Setup()
  messagebox(loSecurity.cErrorMessage)
  return
endif not loSecurity.Setup()

with loSecurity

* Create the roles we'll need.

  .oRoles.Add('Administrators')
  .oRoles.Add('Everyone')

* Create the ADMIN user.

  loUser           = .oUsers.Add('ADMIN')
  loUser.Password  = 'Testing123'
  loUser.FirstName = 'Administrative'
  loUser.LastName  = 'User'

* Create the DHENNIG user.

  loUser           = .oUsers.Add('DHENNIG')
  loUser.Password  = 'DumbPassword'
  loUser.FirstName = 'Doug'
  loUser.LastName  = 'Hennig'

* Add the ADMIN user to the various roles.

  loUser = .oUsers.Item('ADMIN')
  loRole = .oRoles.Item('Administrators')
  .AddUserToRole(loUser.ID, loRole.ID)
  loRole = .oRoles.Item('Everyone')
  .AddUserToRole(loUser.ID, loRole.ID)

* Add the DHENNIG user to the appropriate roles.

  loUser = .oUsers.Item('DHENNIG')
  loRole = .oRoles.Item('Everyone')
  .AddUserToRole(loUser.ID, loRole.ID)

* See which roles are available for DHENNIG, then add
* and remove the Administrators role.
```

```
  loUser      = .oUsers.Item('DHENNIG')
  loAdminRole = .oRoles.Item('Administrators')
  messagebox('DHENNIG is ' + ;
    iif(.IsUserInRole(loUser.ID, loAdminRole.ID), ;
    '', 'not ') + 'in the Administrators role.')
  .AddUserToRole(loUser.ID, loAdminRole.ID)
  messagebox('DHENNIG is ' + ;
    iif(.IsUserInRole(loUser.ID, loAdminRole.ID), ;
    '', 'not ') + 'in the Administrators role.')
  loRoles = .GetRolesForUser(loUser.ID)
  lcRoles = ''
  for each loRole in loRoles
    lcRoles = lcRoles + ;
      iif(empty(lcRoles), '', ', ') + loRole.Name
  next loRole
  messagebox('DHENNIG is in the following roles:' + ;
    chr(13) + chr(13) + lcRoles)
  .RemoveUserFromRole(loUser.ID, loAdminRole.ID)
  messagebox('DHENNIG is ' + ;
    iif(.IsUserInRole(loUser.ID, loAdminRole.ID), ;
    '', 'not ') + 'in the Administrators role.')
endwith
```

Summary

In the first part of this multi-part series on role-based security, we looked at collections of user and role objects and the parts of a security manager class that deals with user and role management. Next month, we'll continue our discussion of SFSecurity, looking at methods that manage the rights a role has to various elements, logging in and logging out users, and encryption and decryption of passwords.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com