Role-Based Security, Part II

Doug Hennig

In part 2 of Doug Hennig's series on role-based security, he looks at code used for manage the rights users and roles have to secured elements, discusses password encryption, and shows how user login works.

Last month, I started a three-part series on role-based security. We looked at the data model for my implementation of security, two classes responsible for maintaining collections of users and roles, and a class called SFSecurity that's responsible for managing the security in an application. This month, we'll continue our examination of SFSecurity, looking at the methods that deal with managing rights, password encryption, and user logging in and out. Please note that I won't show all of the code in these methods; I'll omit parameter checking and error handling code for brevity reasons.

Managing rights

There are several methods that manage the rights users and roles have to the various secured elements in an application. These methods generally accept some or all of these types of parameters:

- User ID, which is the ID of the user's record in the USERS table.
- Element ID, which is the ID for a particular secured element in the ELEMENTS table.
- Role ID: the ID of the role's record in the ROLES table.

The GetUserRightsForElement method finds what rights the specified user has to the specified element. This is probably the most frequently used method in SFSecurity. It's typically used in the SKIP FOR of a menu item to disable items the user doesn't have access to, in a list of reports so only those the user can run appear, or in a form to hide or disable controls for fields the user shouldn't see or edit.

GetUserRightsForElement works by going through the roles the user is in, determining the rights each of them has to the element, and then figuring out the maximum rights of those roles. This is necessary because, for example, a user could be in both the Managers role and the Sales Department role, each of which has different rights to a particular form. While this sounds complex, it only requires a single SQL SELECT statement to determine a user's rights. This method returns the user's rights to the element as a numeric value; it's up to the calling program to interpret this value and decide what to do with it. In this code, cnACCESS_NONE is a constant (defined in SFSECURITY.H, the include file for SFSecurity) that evaluates to 0, meaning no access.

```
lparameters tiElement, ;
  tiUser
local lnRights, ;
  laRights[1]
lnRights = cnACCESS_NONE
select max(RIGHTS) from __SECURITY ;
  where ELEMENT = tiElement and ;
    ROLE in (select ROLE from __USERROLES ;
      where USER = tiUser) ;
  group by ELEMENT ;
  into array laRights
if _tally = 1
  lnRights = laRights[1]
endif _tally = 1
return lnRights
```

UpdateRightsForElement is used to add or edit the rights a specific role has to an element. If there's no record in SECURITY.DBF for the role and element, one is added. Otherwise, the RIGHTS column for the appropriate record is updated with the specified value. This method is usually called from a maintenance form where the user can assign rights for certain elements to specific roles.

```
lparameters tiElement, ;
  tiRole, ;
  tiRights
local llReturn
do case
  case seek(str(tiElement) + str(tiRole), ;
    '__SECURITY', 'SECURITY')
    replace RIGHTS with tiRights in __SECURITY
    llReturn = .T.
  otherwise
    insert into __SECURITY (ELEMENT, ROLE, RIGHTS) ;
      values (tiElement, tiRole, tiRights)
    llReturn = .T.
endcase
return llReturn
```

RemoveRoleForElement removes all access to the specified element for the specified role. It's a very simple method; after all parameter checking and error handling code is stripped out, it consists of a single SQL DELETE statement. Like UpdateRightsForElement, it's usually used in a rights maintenance form.

```
lparameters tiElement, ;
  tiRole
delete from __SECURITY ;
  where ELEMENT = tiElement and ROLE = tiRole
```

Password encryption

Before we discuss logging in users, let's look at how passwords are kept secure. The contents of the PASSWORD field in USERS.DBF obviously must be encrypted or anyone armed something as simple as Notepad can determine the passwords allowing access to the application. So, we need some mechanism to encrypt and decrypt strings. It's rarely a good idea to roll your own code in this area because the chance that your knowledge of security techniques is higher than the bad guys is pretty slim.

Although there are a number of encryption libraries available for VFP, I like a new one called VFPEncryption.FLL, generously provided to the VFP community by new MVP Craig Boyd. It's available, along with sample code demonstrating its use, from Craig's Web site (http://www.sweetpotatosoftware.com/SPSBlog), a resource I highly recommend even if you're not interested in this topic.

To use VFPEncryption.FLL, simply SET LIBRARY TO it (SFSecurity does this in Init), then call the Encrypt or Decrypt functions, passing parameters for the string to operate on, the key to use for encryption, the algorithm to use, and a mode flag (see Craig's documentation for details). To make SFSecurity flexible, I created properties for the latter two values so you can vary them as you see fit. Here's the code from the Encrypt method of SFSecurity (Decrypt is identical but calls the Decrypt function instead).

```
lparameters tcString
return Encrypt(tcString, This.GetKeyValue(), ;
  This.nEncryptionAlgorithm, This.nEncryptionMode)
```

Note that rather than hard-coding the key or putting it into a property, I call a method to return the key value. This is done to prevent a hacker from reading the value of the key from somewhere in memory. GetKeyValue is a protected method, so it can't be accessed from anywhere outside the class, and is purposely written to be rather obtuse to avoid easily determining what it does. It starts with five "seed" strings, stored in the properties cKeySeed1 through cKeySeed5. The code massages these seed values, along with another hard-coded key string, and returns the mangled result. Thanks to Christof Wollenhaupt and Tamar Granor for the idea for this code.

```
#define cnKEY ":E9}$6+a7ifN)#G')/&[ZpM$4Z.JOy@M" + ;
  "&{^%JFOd2Ew{A?=dxB]RB9un"
local lcSeed, ;
  lcKey, ;
  lnI, ;
  lcChar, ;
  lnChar
```

```
lcSeed = This.cKeySeed5 + This.cKeySeed1 + ;
  This.cKeySeed3 + This.cKeySeed4 + This.cKeySeed2
lcKey  = ''
for lnI = 1 to len(lcSeed)
  lcChar = substr(lcSeed, lnI, 1)
  lnChar = asc(lcChar)
  do case
    case mod(lnI, 5) = 0
      lcKey = lcKey + chr(abs(lnChar - lnI))
    case mod(lnI, 4) = 0
      lcKey = lcKey + lcChar
    case mod(lnI, 3) = 0
      lcKey = lcKey + chr(mod(lnChar * 2, 255))
    case mod(lnI, 3) = 0
      lcKey = lcKey + substr(cnKEY, ;
        mod(lnChar, 56), 1)
    case isalpha(lcChar)
      lcKey = lcKey + iif(mod(lnI, 2) = 0, ;
        lower(lcChar), upper(lcChar))
    otherwise
      lcKey = lcKey + chr(mod(lnChar + 65, 255))
  endcase
next lnI
do case
  case This.nEncryptionAlgorithm = cnENCRYPT_AES192
    lcKey = left(lcKey, 24)
  case This.nEncryptionAlgorithm = cnENCRYPT_AES256
    lcKey = left(lcKey, 32)
  case This.nEncryptionAlgorithm = ;
    cnENCRYPT_BLOWFISH448
    lcKey = left(lcKey, 56)
  otherwise
    lcKey = left(lcKey, 16)
endcase
return lcKey
```

Note that the password for a user is stored in a user object (see last month's article for a discussion of user objects) as the same encrypted value in the table. That's done for the same reason discussed earlier: to prevent someone from reading clear text password values in memory. In fact, nowhere in SFSecurity is a decrypted password ever stored in a variable or a property. Notice the code in Encrypt shown earlier simply returns the result of the Encrypt function rather than storing it somewhere first. Notice also the code in ValidateLogin (we'll discuss this method in more detail later), which encrypts the password entered by the user when logging in before comparing it to the stored password:

```
llReturn = loUser.Password==This.Encrypt(tcPassword)
```

These steps help minimize the ability of bad guys determining passwords by simply looking in memory for their decrypted values.

User log in and log out

Now that we have methods to manage roles, users, and rights, let's look at how we identify the user. The LogIn method supports multiple login mechanisms, determined with the setting of the nLoginMethod property. One mechanism is programmatic login: pass this method the user name and password and, if they're valid, that user is logged in. Another mechanism uses the user's Windows login; in that case, we won't bother with a password, the idea being that if they've successfully logged into Windows, we'll let them into our application as well.

A third login mechanism is for automatic logging in. This is normally only used in a development environment, and is a tip I learned from the much missed Drew Speedie. Why force the developer to log in each time they test the application? Instead, automatically log them in using the name stored in cAutoUserName (set to ADMIN by default but could be any other valid user name).

The final mechanism, the one that's normally used, is to display a dialog where the user can enter their user name and password. The name and library for the class to use as the login dialog are stored as properties (cLoginFormClass and cLoginFormLibrary) rather than hard-coded because you may wish under some conditions to use a different dialog. For example, if they can access more than one database, you may wish to have an additional control on the login form asking them to select the database to log into.

Regardless of the mechanism, once we have the user name, we find that user in the users collection and set the oCurrentUser property to the user object for that user. If something went wrong during the login process, the cErrorMessage property is set to the appropriate error message.

```
lparameters tcUserName, ;
  tcPassword
local lcMessage, ;
  lcUserName, ;
  loForm, ;
  loUser, ;
  llReturn
with This

* Log out the current user, then log in depending on
* the chosen mechanism.

  .LogOut()
  do case

* A user name and password were specified, so see if
* they're valid.

    case not empty(tcUserName) and ;
      .ValidateLogin(tcUserName, tcPassword)
      lcUserName = tcUserName

* They aren't valid, so get the appropriate error
* message.

    case not empty(tcUserName)
      .cErrorMessage = ;
        .GetLocalizedString('ERR_INVALID_LOGIN')

* Get the user's Windows login name if we're supposed
* to.

    case .nLoginMethod = cnUSER_LOGIN_WINDOWS
      lcUserName = .GetUserFromWindows()

* Use the cAutoUserName property if we're
* automatically logging in.

    case .nLoginMethod = cnUSER_LOGIN_AUTO
      lcUserName = .cAutoUserName

* Display a login dialog if we're supposed to.

    case .nLoginMethod = cnUSER_LOGIN_FORM
      loForm = newobject(.cLoginFormClass, ;
        .cLoginFormLibrary, '', This)
      loForm.oLocalizer =.oLocalizer
      loForm.Show()
      if vartype(loForm) = 'O'
        lcUserName = alltrim(loForm.cUserName)
      endif vartype(loForm) = 'O'
  endcase
  do case

* We have a user name, so find the user in the user
* collection. If we can't, store the error in
* cErrorMessage.

    case not empty(lcUserName)
      loUser = .oUsers.Item(lcUserName)
      if vartype(loUser) = 'O'
        .oCurrentUser = loUser
        llReturn      = .T.
      else
        lcMessage = ;
          .GetLocalizedString('ERR_USER_NOT_FOUND')
        .cErrorMessage = strtran(lcMessage, ;
          ccMSG_INSERT1, lcUserName)
      endif llIsValidUser

* If we don't have a user name, display an error if
* we're not logging in from a form.

    case .nLoginMethod <> cnUSER_LOGIN_FORM
      .cErrorMessage = ;
        .GetLocalizedString('ERR_NO_USER_SPECIFIED')
  endcase
endwith
return llReturn
```

The LogOut method is very simple: it just sets oCurrentUser to NULL. If oCurrentUser doesn't contain the user object for the currently logged-in user, there is no logged-in user.

The GetUserFromWindows method uses a Windows API function to determine the user's Windows login name.

```
local lcName, ;
  lnSize, ;
  lcUser, ;
  lnStatus, ;
  lcUserName

* Initialize the variables we need and declare the
* Windows function.

lcName = ccNULL
lnSize = cnUSER_BUFFER_SIZE
lcUser = replicate(lcName, lnSize)
declare integer WNetGetUser in Win32API ;
  string@ cName, string@ cUser, integer@ nBufferSize

* Call the function and return the result.

lnStatus   = WNetGetUser(@lcName, @lcUser, @lnSize)
lcUserName = iif(lnStatus = 0, ;
  upper(strtran(left(lcUser, lnSize), ccNULL)), '')
return lcUserName
```

ValidateLogin determines whether the specified user name and password are valid. It tries to find the specified user in the users collection and, if found, checks whether the password matches. As discussed earlier, it encrypts the specified password and compares the result to the encrypted password for the user so the unencrypted password is never stored in a memory variable.

```
lparameters tcUserName, ;
  tcPassword
local loUser, ;
  llReturn
with This
  loUser = .oUsers.Item(tcUserName)
  if vartype(loUser) = 'O'
    llReturn = loUser.Password==.Encrypt(tcPassword)
  endif vartype(loUser) = 'O'
endwith
return llReturn
```

The login dialog shown in **Figure 1** is defined in SFLoginForm, in SFSecurityUI.VCX. (I'm in the habit now of separating UI from non-UI classes, even if they have related functions, so I can exclude the UI class libraries from projects used to build COM objects. There's no point in making the DLL larger than necessary by including classes that will never be used.) This class isn't very complicated: it consists of a couple of labels, a couple of textboxes (txtPassword has its PasswordChar property set to "*" so it displays asterisks instead of text), and a couple of buttons. The Click method of cmdOK calls the Validate method of the form to validate the user name and password; if it returns .T., Click calls the Hide method of the form to return control to the caller but not release the form just yet because the caller will want to retrieve the entered user name from the form.

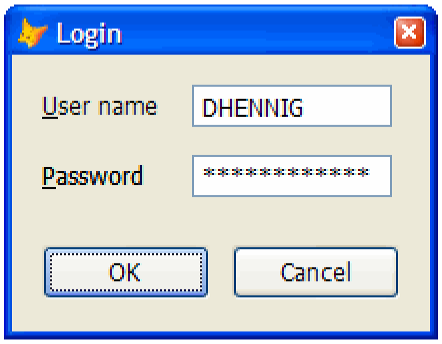

Figure 1. The SFLoginForm class is a dialog in which the user can log in.

The Validate method of the form calls the ValidateLogin method of SFSecurity (a reference to which is stored in the oSecurity property) to test if the user name and password entered by the user are valid, and if so, sets the lValidUser property to .T.

```
local lcUser, ;
  lcMessage
with This
  lcUser = trim(.txtUserName.Value)
  if not empty(lcUser)
    .lValidUser = .oSecurity.ValidateLogin(lcUser, ;
      trim(.txtPassword.Value))
    if .lValidUser
      .cUserName = lcUser
    else
      lcMessage = ;
        .GetLocalizedString('ERR_INVALID_LOGIN')
      messagebox(lcMessage, MB_ICONEXCLAMATION, ;
        .GetLocalizedString('FRM_CAP_LOGIN_FAILED'))
      .txtPassword.SetFocus()
    endif .lValidUser
  endif not empty(lcUser)
endwith
return This.lValidUser
```

Check it out

To see SFSecurity in action, run MAIN.PRG. It creates an SFSecurity object, sets the password encryption seed values, ask you to log in, and then displays a menu. The menu, SAMPLE.MNX, calls GetUserRightsForElement in the Skip For clause of each bar so only authorized users can access the function. Log in as DHENNIG and notice which functions you have access to and what rights you have when you choose them. Exit, run it again, and log in as ADMIN this time, and notice the difference in rights this user has.

Summary

That's it for this month. We finished looking at SFSecurity, discussing the methods used for managing rights, password encryption, and user log in, as well as SFLoginForm, a simple login dialog. Next month, we'll finish our examination of my role-based security implementation by looking at a dialog used to maintain users, roles, and the rights roles have to particular secured objects.

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 the world. 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