

# Application Security

*Doug Hennig*

**Most applications need a way to limit which users can run an application and which functions users can perform in the application. This article presents a set of tools for defining and maintaining application security. It also pulls together the reusable tools described in previous issues to provide a simple application framework.**

A feature most applications require is the ability to prevent some users from accessing some functions, or limiting what capabilities certain users have in certain forms or programs. A simple example is a Reindex function; you only want users who know what they're doing to choose this function, or the application could be tied up for hours just because someone wondered what that was for.

I'm not talking about robust, mainframe style security. After all, we're working with DBF files which anyone who knows what they're doing could get into and muck about with. To provide true data security, you'd have to use third party encryption tools that prevent access to the data outside an application. What I'm referring to is application security, which limits what users can run the application and what functions they can do in the application. This article describes a set of tools that provide a simple security system for your applications. These tools were designed to be extendible so they can cover more types of security than the current implementation provides

The sample application available on the Subscriber Download site includes not only the security tools described in this article, but several other tools I've written about over the past several months. This application ties several of these pieces together to show how they collaborate to provide common services to an application framework. For example, the SFErrorMgr class described in the January and February 1998 issues provides error handling services. The SFOpen form and associated classes described in the November 1997 issue provide a File Open dialog. SFMaintForm (for data entry forms), SYSMAIN.MNX (application menu), SFApplication (application manager), and SYSMAIN.PRG (application startup) were provided (but not described in much detail) with the error handling class as a simple application framework.

## Security Model

The security model used by the tools provided in this article is as follows:

- There are three levels of security. The first level is application: the user can either run the application or not. The second level is object: the user can either access a security object (anything that needs to be secured, such as a form, PRG, field, etc.) within the application or not. The third level is function: what rights does the user have to manipulate a certain object (for example, can the user add records in a form)?
- Currently, the only type of security object supported is "module" (anything the application can run: a form, a PRG, an external utility, etc.). However, support for other types of objects (such as reports or fields) could be added.
- The only functions supported are "full access", "add", "edit", "delete", and "view". You could easily provide additional functions, since the function is simply represented by a value in a field in a security table. For example, for field level security, you could make certain fields (such as salary information) read-only or invisible to users based on their security settings.
- A user must be identified to the application. This might be by having them enter their name and password in a login screen, or it might be done automatically by asking Windows for their user name. An oUser object, instantiated from the SFUser class, provides this service via its Login() method, and manages other aspects of the user as well.
- A USERS table keeps a list of users authorized to access the application. It could also keep user-specific preferences if desired, since once a user is identified to the application, the oUser object can have properties indicating user preferences. The SFUser class doesn't have any preferences properties, but you could subclass it, add any properties desired, and put code in the GetCustomUserProperties() method (which is empty in SFUser) to populate these properties. GetCustomUserProperties() is called

from the Login() method after the user has successfully logged in and the USERS table is positioned to the record for the user.

- The MODULES table described in the November 1997 issue of FoxTalk (“A File Open Dialog”) provides details on which modules are available in the application and what the security requirements of each are. Three types of module security are provided: accessible to all users (no security); accessible or not, but having no additional functional security; or accessible plus functional (“add, edit, or delete”) security.
- A SECURITY table is used to resolve the many-to-many relationship between USERS and MODULES. It indicates which modules a certain user can access and what functional rights they have in those modules. Access is determined by the existence of a record; if there is no record for a certain user for a particular module, that user cannot access that module.

Details of this model are described in more detail in the rest of this article. Due to space limitations, not much code will be shown in this article. For code details, download the files for this article from the Subscriber Downloads site.

### Users Table

The users defined to the application (that is, those who have access to it) are defined in the USERS table. Tables 1 and 2 show the structure of this table.

*Table 1. Structure of USERS.DBF.*

Field	Type	Size	Description
USERNAME	Character	10	The user's login name
PASSWORD	Character	10	The user's password
LASTNAME	Character	25	The last name of the user
FIRSTNAME	Character	15	The first name of the user
PHONE	Character	8	The user's phone number
SUPERVISOR	Logical	1	.T. if the user is a "supervisor"
DEVELOPER	Logical	1	.T. if the user is a developer

*Table 2. Indexes for USERS.DBF.*

Tag	Type	Expression
FULLNAME	Regular	upper(LASTNAME + FIRSTNAME)
USERNAME	Candidate	USERNAME
DELETED	Regular	deleted()

The user's password is stored as an encrypted string. Encryption and decryption are performed by ENCRYPT.PRG and DECRYPT.PRG. These provide pretty simple encryption; substitute your own routines if you need a more robust algorithm. If SUPERVISOR is .T., this user is considered to be a “super-user”, and they have all rights to all modules. DEVELOPER is used to provide run-time developer features; if it's .T., a developer menu is shown (defined in PROGTOOL.MNX) that includes a run-time “suspend” function (CMDHELL.PRG), and error messages include a “Debug” option (see the January 1998 issue of FoxTalk for information on the SFErrorMgr class).

### Modules Table

The modules used by the application are defined in the MODULES table, which is the source of information for both the File Open dialog and the security requirements of each module. Tables 3 and 4 show the structure of this table; see the “A File Open Dialog” article in the November 1997 issue of FoxTalk for the non-security aspects of this table.

*Table 3. Structure of MODULES.DBF.*

Field	Type	Size	Description
NAME	Character	40	The name of the module as the user sees it
MODULE	Character	40	The name of the module as the application knows it (e.g. form name)
DESCRIP	Memo	4	A full description of the module

GROUP	Character	40	The group the module belongs to
HOWTOCALL	Character	45	A string that when macro expanded will execute the module
SECURITY	Numeric	1	The type of security used for this module
INOPENFORM	Logical	1	.T. if this module should appear in the File Open dialog
ORDER	Numeric	3	The order the group should appear in the dialog
IMAGEKEY	Character	20	The name of the ICO file used as the image for this module

*Table 4. Indexes for MODULES.DBF.*

Tag	Type	Expression
MODULE	Regular	upper(MODULE)
ORDER	Regular	str(ORDER) + upper(GROUP + NAME)
NAME	Regular	upper(NAME)

SECURITY contains one of three values: 0 for no security (all users can access the module), 1 for “yes/no” security (meaning the user can either access the module or not), or 2 for modules that support “read/write” security (some users can add, edit, or delete records while others can only view them).

Since MODULES doesn’t change at run-time, it should be included in the project so it’s built into the EXE.

### Security Table

The SECURITY table indicates what rights each user has in each module. Tables 5 and 6 show the structure of this table.

*Table 5. Structure of SECURITY.DBF.*

Field	Type	Size	Description
USERNAME	Character	10	The user’s login name
TYPE	Character	1	The type of security object
MODULE	Character	40	The name of the security object
ACCESS	Character	3	The rights the user has to this security object

*Table 6. Indexes for SECURITY.DBF.*

Tag	Type	Expression
SECURITY	Regular	upper(USERNAME + TYPE + MODULE)
DELETED	Regular	deleted()

The TYPE field indicates the type of security object. Currently, only modules (“M”) are supported, but you could implement additional types of objects using different type values (for example, “F” for fields, in which case MODULE would contain the name of the field). ACCESS contains “F” for full rights, “A” for add rights, “E” for edit rights, “D” for delete rights, or “V” for view-only rights. These values aren’t all mutually exclusive; a user with add and edit rights would have “AE” in this field. Additional types of rights could be handled by using additional values.

### SFUser

SFUser (defined in SFAPPLIC.VCX) is the user manager class. This class is based on SFCustom (our Custom class defined in SFCTRLS.VCX). It uses USER.H as its include file so constants can be used for readability purposes. Its Init() method accepts two optional parameters: the name of the application INI file (this is only used if the user will be logged in from information contained in the INI file as we’ll see in a moment) and the directory where the USERS and SECURITY tables can be found (if it isn’t passed, these tables should be in the current directory or FoxPro path). The OpenUserTables() method opens the USERS and SECURITY tables if they aren’t already open; this method is called from any method needing to access these tables.

The Login() method logs in a user and stores the user name in the cUserName property. It also sets the IDeveloper and ISupervisor properties based on the DEVELOPER and SUPERVISOR fields in the USER table. Login() uses one of four login mechanisms, based on the setting of the nLoginMethod property:

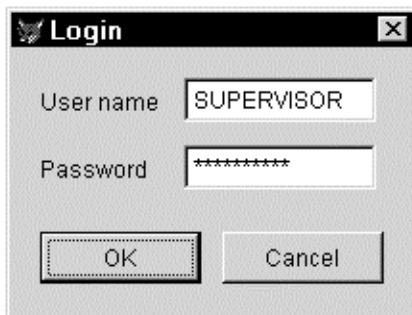
- If nLoginMethod is 1, the GetUserFromINI() method is called. This method checks the environment variable key (the name of the key is defined in the cEnvVariableKey property) of the user section (the name of the section is defined in the cINISection property) of the application INI file (defined in the cINIFile property) for the name of a DOS environment variable that contains the user name. Say APPLIC.INI contains the following:

```
[User]
EnvVariable = USERNAME
```

In this case, GetUserFromINI() expects there's a DOS environment variable called USERNAME that contains the name of the logged in user. This is commonly used with Novell networks, which are often configured to store the user name in a DOS environment variable.

- If nLoginMethod is 2, the GetUserFromWindows() method is called. This method uses the Windows API function WNetGetUser to get the name of the current user.
- Login automatically logs in the user when nLoginMethod is 3, assigning the user name stored in the cAutoUserName property. This mechanism is usually only used during development. After all, why go through the hassle of logging in every time you run the application in a test environment? If you set nLoginMethod to 3 and cAutoUserName to the name of the user you want to log in as (the default is "SUPERVISOR"), you'll automatically be logged in as that user each time.
- The user is asked to log in using the SFLogin form shown in Figure 1 if nLoginMethod is 4. SFLogin is quite simple: it asks for the name of the user and the password (using an SFPassword textbox, defined in SFCCTRLS.VCX, which has the PasswordChar property set to "\*"), and only enables the OK button if that user name can be found in the USERS table and the password is correct.

Figure 1. The SFLogin form is used to log users in.



After Login() determines what the user name is, it then checks for that name in the USERS table. If it isn't found, an error message is displayed and Login() returns .F.; the calling program should terminate the application under these conditions. If the user name was found in USERS, Login() then calls the abstract method GetCustomUserProperties(). This method is intended to be used in a subclass of SFUser to populate custom properties of the subclass with user preferences stored in fields you add to the USERS table. For example, say you want to store the name of a form to automatically run when the user logs in. You could create a field in USERS called FORM, create a custom property of a subclass of SFUser called cForm, and put the following code into the GetCustomUserProperties() method:

```
This.cForm = USERS.FORM
```

The CheckAccess() method is used to determine if the current user can access a particular security object. This method accepts two parameters: the name of the security object and the type of the object (optional: if it isn't specified, "M" for module is used). It returns either blank if the user has no rights to the specified object or the value of SECURITY.ACCESS for the record for that object and the current user. If the user is a supervisor (oUser.ISupervisor is .T.), it returns "F" (full rights). This method can be used

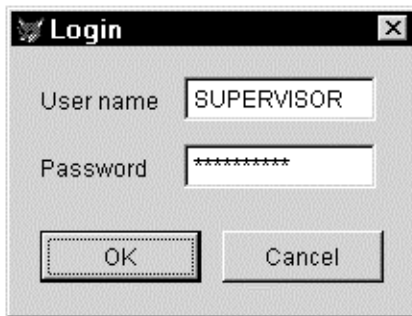
anywhere you need to determine if a user can access a security object (such as in the SKIP FOR clause of a menu item) or what rights the user has (such as in a data entry form to determine if the user can add new records). If desired, you could use Andrew Ross MacNeill's GENMENUX to remove bars the user doesn't have rights to in a menu rather than disabling them.

For an example of how CheckAccess() is used to disable menu items, see the SKIP FOR clause of the Security function in the Utilities pad of SYSMAIN.MNX in the sample code found on the Subscriber Downloads site. For an example of how it can be used to determine what rights the user has in a form, see the Init() method of TEST.SCX.

## Maintaining Security

Application security is maintained using the SF Secur form. This form, which is based on the data entry form class SFMaintForm defined in SFFORMS.VCX (which we won't go over in this article), is shown in Figure 2. SF Secur is itself a security object, so it has a record in the MODULES table, and only certain users (such as an application supervisor) can access it. The supervisor can define which users have access to the application by adding a record for the user to the USER table using this form. If the Supervisor checkbox is checked, the user has full access to everything in the application, because the SFUser CheckAccess() method returns .T. for supervisor users.

Figure 2. The SF Secur form is used to maintain application security.



Below the user information area of the form is a listbox containing the modules requiring security defined in the MODULES table (those with a SECURITY value greater than 0). To the right of the listbox is a set of controls used to define what rights this user has in the selected module. If MODULES.SECURITY is 1, the module only needs access rights (the user can either run the module or not), so "No Access" and "Full Access" are the only options. If MODULES.SECURITY is 2, the module supports "add/edit/delete" rights, so a "Partial Access" option also appears, and check boxes for Add, Edit, and Delete appear. If Partial Access is selected but none of the check boxes are checked, the user has view-only rights in the module. The Save() and Cancel() methods of the form (called from the File Save and File Revert items in the sample SYSMAIN.MNX) save or revert changes to the USERS and SECURITY records.

The DataEnvironment of SF Secur contains the USERS, SECURITY, and MODULES tables. However, since the location of USERS and SECURITY might be different at run-time than design time (MODULES is built into the EXE, so we don't need to worry about it), and the oUser object knows where these tables are located, the BeforeOpenTables() method of the DataEnvironment calls GETUDATA.PRG, which simply sets the CursorSource property of the USERS and SECURITY cursors to the appropriate values found in the oUser object.

The SFChgPw form allows the user to change their password. It has three SFPassword fields. The first one requires the user to enter the existing password; if they don't enter the correct password, they can't change it. The other two are used to enter the new password twice (the second time for confirmation). Like SF Secur, its DataEnvironment.BeforeOpenTables() method calls GETUDATA.PRG to ensure the USERS table can be found.

## Example

Let's take a look at the sample application on the Subscriber Download site. The MYAPP project contains all the pieces used in this simple application. USERS.DBF initially contains two records, one for a user called SUPERVISOR (with "SUPERVISOR" as the password) and one for GUEST (with no password).

SYSMAIN.PRG is the main program. It instantiates several objects (the SFApplication, SFEnvironment, and SFErrorMgr classes first provided in January 1998's error handler article), and then instantiates the MyUser class. MyUser is defined programmatically at the end of SYSMAIN.PRG as a subclass of SFUser. Its Init() method asks you what mechanism you'd like to use for logging in. Obviously, this wouldn't be used in a production application, but is useful for easily trying out the various ways of logging in. When asked what mechanism you'd like, enter a value between 1 and 4 (see above for how this value determines the mechanism). The main program then calls the Login() method of the oUser object; only if it succeeds, meaning the user successfully logged in, does the rest of the application startup continue.

After the user logs in, the IShowDebug and cUser properties of the oError object (described in January 1998's article) are set to the IDeveloper and cUserName properties of the oUser object so the error handler knows who the user is (for error logging purposes) and if "debug" should be available as an option when it displays an error message. The application's menu is then displayed, and if the user is a developer, a special developer's menu (defined in PROGTOOL.MNX) is displayed. This menu provides quick access to the Watch, Trace, and View windows, a hot key for putting a reference to the object the mouse is over into the global variable O (handy for determining its name or other properties), and a pseudo-Command window (CMDHELL.PRG) for run-time environments (extremely useful for executing small instructions when a development copy of VFP isn't available, such as at a client site).

The application menu has a File Open function that displays the File Open dialog described in November 1997's article. Depending on which user you logged in as, you may not see all the available forms in this dialog, since the GUEST user doesn't have access to the Clients form (unless you change it using the Security form). Also, the GUEST user can't access the Security form, so the Security function in the Utilities menu is disabled.

Log in as SUPERVISOR (either by using login mechanism 3 or using mechanism 4 and entering SUPERVISOR for both the user name and password in the login form). Choose the Security function and add a record for your own user name. Give yourself whatever rights you'd like to try out, then exit the application and rerun it. Log in with mechanism 1 (if you use the DOS environment variable approach; edit APPLIC.INI to contain the name of the appropriate environment variable) or 2 (if you log in to Windows) and see which functions you can access.

## Summary

We're starting to see some synergy between the reusable tools I've presented over the past several months. The user object shares information with the error handling object so the name of the current user can be logged when errors occur. The application object displays a developer menu if the user object says this is a developer user. The File Open dialog only displays modules the current user has rights to. You may not have noticed it, but we've slowly been building the pieces of a simple application framework. We'll continue to add to this framework and refine it further over the coming months.

*Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, spoke at the 1997 Microsoft FoxPro Developers Conference (DevCon), and will be speaking at the 1998 DevCon. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.*