Keep Y’er Paws Off My Stuff

Doug Hennig

Need to prevent unauthorized access to your FoxPro data? Cryptor from Xitech provides an easy way to encrypt files and leave them encrypted on disk, even while you access them as normal tables in your application.

A common question on VFP online forums is how to protect data from unauthorized users. There are several alternatives:

- Upsizing to a database engine that natively provides security such as SQL Server or Oracle. While this may be a good long-term strategy (or not, depending on your viewpoint), it may mean a major redesign of your application.
- Encrypting the data in a table. Although you could roll your own encryption and decryption routines, a better choice would be to use the crypto functions built into Windows; although they can be complicated to use, VFP comes with a wrapper class in the FoxPro Foundation Classes (_CryptAPI in _Crypt.VCX) to make them more approachable. However, this approach is difficult to add to an application, because everything that touches the table must decrypt data when reading it and encrypt it when writing it.
- Encrypting a table on disk, decrypting it when your application opens it, and decrypting it again when you’re done with it. Like the previous alternative, this one is complicated by the fact that everything accessing the table must perform these tasks. It also leaves the table open to unauthorized access while it’s open in your application since it resides in an unencrypted state during that time, plus has issues with multi-user concurrency.

Fortunately, there’s a better solution: Cryptor from Xitech. This library provides functions to encrypt and decrypt strings and files, and even better, can leave the files encrypted on disk even while your application accesses them.

Installing Cryptor

You can download an evaluation version of Cryptor from http://www.xitech-europe.co.uk. The main file is XICRCORE.DLL; this has the functions that provide Cryptor’s services. There are also several FLL files for VFP developers that provide a wrapper for functions in XICRCORE. However, because they’re version specific (for example, C40FOX70.FLL is for VFP 7), mean at least one more file to ship with your application, and using XICRCORE functions isn’t difficult, they really aren’t necessary in my opinion. Cryptor also comes with a help file and support files for use with Delphi, C++, and Visual Basic.

Using Cryptor

Because Cryptor functions are contained in a DLL, you have to declare them before you can use them. Here’s some code (adapted from TESTAPI.PRG in this month’s downloads) that declares the functions we’ll use in XICRCORE.DLL:

```
lcDLL = 'XICrCore.DLL'
declare integer CRYIni_Initialize      in XICrCore.DLL ;
  integer LoadMode
declare integer CRYIni_UnInitialize    in XICrCore.DLL
declare integer CRYUtl_EncodeString    in XICrCore.DLL ;
  string strSrc, string @ strDest, integer dwLength, ;
  string strPassword, integer dwMethod
declare integer CRYUtl_DecodeString    in XICrCore.DLL ;
  string strSrc, string @ strDest, integer dwLength, ;
  string strPassword, integer dwMethod
declare string  CRYUtl_GetErrorMessage in XICrCore.DLL ;
  integer errorCode
declare integer CRYUtl_Encode          in XICrCore.DLL ;
  string strFilename, string strPassword, ;
  string strBackupExt, integer bKeepBackup, ;
  integer dwMethod
declare integer CRYUtl_Decode          in XICrCore.DLL ;
  string strFilename, string strPassword, ;
  string strBackupExt, integer bKeepBackup, ;
  integer dwMethod
declare integer CRYMan_Register        in XICrCore.DLL ;
  string strFilename, string strPassword, ;
  integer dwFlags, integer dwMethod
declare integer CRYMan_Unregister      in XICrCore.DLL ;
  string strFilename
```

Before you can do anything with Cryptor, you have to initialize it using the CRYIni_Initialize function. This function must be passed a parameter indicating the "load mode" (the Cryptor help file explains which values to use and why); in the following and other code in this article, CRYPTOR_LOADMODE_NORMAL and other upper-cased things are constants defined in SFCRYPTOR.H. Like most other Cryptor functions, CRYIni_Initialize returns a numeric value indicating whether it succeeded (0, or the constant CRYPTOR_ERR_SUCCESS) or, if not, what the error code is. In this code, we'll accept either CRYPTOR_ERR_SUCCESS or CRYPTOR_ERR_ALREADY_INITIALIZED (indicating that Cryptor has already been initialized) as meaning success.

```
lnStatus = CRYIni_Initialize(CRYPTOR_LOADMODE_NORMAL)
if not inlist(lnStatus, CRYPTOR_ERR_SUCCESS, ;
  CRYPTOR_ERR_ALREADY_INITIALIZED)
  messagebox('Could not initialize Cryptor: status ' + ;
    'code ' + transform(lnStatus))
  return
endif not inlist(lnStatus ...
```

Once you're finished using Cryptor, you should uninitialize it:

```
lnStatus = CRYIni_UnInitialize()
```

Encrypting a string is easy: simply call the CRYUtl_EncodeString function, passing it the string to encrypt, a string (passed by reference) to store the encrypted value into, the length of the string, the password to use for encryption, and the method of encryption to use. Cryptor provides several levels of encryption; the help file only briefly mentions the differences in these levels, likely for trade secret reasons. If the encryption process fails, you can call CRYUtl_GetErrorMessage to get the text of the error message (although the help file states that CRYUtl_EncodeString should always return CRYPTOR_ERR_SUCCESS). Here's an example using level 2 encryption:

```
lcString    = 'MyString'
lnLen       = len(lcString)
lcEncrypted = space(lnLen)
lcPassword  = 'FoxRocks!'
lnMethod    = ENCODER_LEVEL2
lnStatus    = CRYUtl_EncodeString(lcString, ;
  @lcEncrypted, lnLen, lcPassword, lnMethod)
if lnStatus = CRYPTOR_ERR_SUCCESS
  messagebox(lcString + ' encrypts using method ' + ;
    transform(lnMethod) + ' and password ' + ;
    lcPassword + ' to ' + lcEncrypted)
else
  messagebox('Could not encrypt string: ' + ;
    CRYUtl_GetErrorMessage(lnStatus))
endif lnStatus = CRYPTOR_ERR_SUCCESS
```

Decrypting is just as easy, and uses almost identical code. Here, we'll simply decrypt the string previously encrypted to make sure we can get it back again.

```
lcString = space(lnLen)
lnStatus = CRYUtl_DecodeString(lcEncrypted, ;
  @lcString, lnLen, lcPassword, lnMethod)
```

```
if lnStatus = CRYPTOR_ERR_SUCCESS
  messagebox('The encrypted string decrypts back to ' + ;
    lcString)
else
  messagebox('Could not decrypt string: ' + ;
    CRYUtl_GetErrorMessage(lnStatus))
endif lnStatus = CRYPTOR_ERR_SUCCESS
```

In addition to strings, you can also encrypt and decrypt entire files using the CRYUtl_Encode and CRYUtl_Decode functions. Pass these functions the fully-qualified name of the file, the password to use, the extension to use for a backup file Cryptor creates during the process (.NULL. means use the default CBK extension), 0 to delete the backup after completion or 1 to retain it, and the encryption method to use. Here's an example that encrypts a table, tries to open it (which should fail because it's no longer recognizable as a VFP table), then decrypts it and tries to open it again (which should succeed).

```
create table TEST (FIELD1 C(10), MEMO1 M)
index on FIELD1 tag FIELD1
insert into TEST values ('A', 'My memo')
use
lcFile   = fullpath('TEST.DBF')
lnStatus = CRYUtl_Encode(lcFile, lcPassword, .NULL., 0, ;
  lnMethod)
if lnStatus = CRYPTOR_ERR_SUCCESS
  messagebox('We should get an error when trying to ' + ;
    "USE the table because it's encrypted. Choose " + ;
    'Ignore so we can continue.')
  use TEST
  lnStatus = CRYUtl_Decode(fullpath('TEST.DBF'), ;
    lcPassword, .NULL., 0, lnMethod)
  if lnStatus = CRYPTOR_ERR_SUCCESS
    use TEST
    if used('TEST')
      messagebox('The table was successfully decrypted.')
      browse
      use
    endif used('TEST')
  else
    messagebox('Could not decrypt table: ' + ;
      CRYUtl_GetErrorMessage(lnStatus))
  endif lnStatus = CRYPTOR_ERR_SUCCESS
else
  messagebox('Could not encrypt table: ' + ;
    CRYUtl_GetErrorMessage(lnStatus))
endif lnStatus = CRYPTOR_ERR_SUCCESS
```

All of this is fine, but where Cryptor really shines is the ability to decrypt a file only in memory. With other mechanisms that encrypt a file, you have to decrypt it before you can use it. That means while it's being used, it's susceptible to prying eyes. With Cryptor's CRYMan_Register function, you specify that certain files are to remain encrypted on disk but should be automatically decrypted in memory as they're used in an application. Thus, you get protection from unauthorized access to the files because they're never decrypted on disk.

To register one or more files with Cryptor, pass the CRYMan_Register function the name of file (you can use wild cards, such as MYTABLE.*, to register multiple files with one call), the password, 0 (this parameter isn't used; it's for backward compatibility with earlier versions of Cryptor), and the encryption method. This example assumes TEST.DBF, CDX, and FPT are encrypted, so it registers them with Cryptor so VFP can open the table. It then uses the CRYMan_Unregister function to unregister them; VFP will no longer be able to open the table.

```
lcFile   = fullpath('TEST.*')
lnStatus = CRYMan_Register(lcFile, lcPassword, 0, ;
  lnMethod)
if inlist(lnStatus, CRYPTOR_ERR_SUCCESS, ;
  CRYPTOR_ERR_ALREADY_REGISTERED)
  use TEST
  if used('TEST')
    messagebox('The table was decrypted in memory ' + ;
      'so we can use it.')
    browse
    use
  endif used('TEST')
  lnStatus = CRYMan_Unregister(lcFile)
  if lnStatus = CRYPTOR_ERR_SUCCESS
    messagebox('We should get an error when trying ' + ;
      "to USE the table because it's still " + ;
      'encrypted. Choose Ignore so we can continue.')
    use TEST
  else
    messagebox('Could not unregister table: ' + ;
      CRYUtl_GetErrorMessage(lnStatus))
  endif lnStatus = CRYPTOR_ERR_SUCCESS
else
  messagebox('Could not register table: ' + ;
    CRYUtl_GetErrorMessage(lnStatus))
endif inlist(lnStatus ...
```

SFCryptor

Although using Cryptor is easy, there are functions to declare, initialization and uninitialization to be done, the same parameters to pass to each method, and so forth. I created a wrapper class called SFCryptor, based on SFCustom (my Custom base class in SFCTRLS.VCX), that handles all the messy details. For example, here’s some code (adapted from TESTSFCRYPTOR.PRG) that encrypts and decrypts a string. Notice you don’t have to pass the password and encryption method with every call or do any initialization or other work.

```
loCryptor = newobject('SFCryptor', 'SFCryptor.vcx')
loCryptor.nEncryptionMethod = ENCODER_LEVEL2
loCryptor.cPassword         = 'FoxRocks!'

* Encrypt a string.

lcString    = 'MyString'
lcEncrypted = loCryptor.EncryptString(lcString)
if not isnull(lcEncrypted)
  messagebox(lcString + ' encrypts using method ' + ;
    transform(loCryptor.nEncryptionMethod) + ;
    ' and password ' + loCryptor.cPassword + ' to ' + ;
    lcEncrypted)

* Decrypt the same string.

  lcString = loCryptor.DecryptString(lcEncrypted)
  if not isnull(lcString)
    messagebox('The encrypted string decrypts back ' + ;
      'to ' + lcString)
  else
    messagebox(loCryptor.cErrorMessage)
  endif lnStatus = CRYPTOR_ERR_SUCCESS
else
  messagebox(loCryptor.cErrorMessage)
endif not isnull(lcEncryptedString)
```

The EncryptString and DecryptString methods are very simple: they just call the protected ProcessString method. Here’s the DecryptString method (EncryptString is identical except it passes “encrypt” to ProcessString). Notice that it can accept password and encryption method parameters; if you don’t pass them, the values in the cPassword and nEncryptionMethod properties are used.

```
lparameters tcString, ;
  tcPassword, ;
  tnMethod
return This.ProcessString(tcString, tcPassword, ;
  tnMethod, 'decrypt')
```

ProcessString starts by ensuring a string was passed; it uses the new EVL() function added in VFP 8 to check the tcString parameter without having to worry about the data type (if you want to use this class with earlier versions of VFP, replace this statement with one that uses VARTYPE() to ensure a string was passed). It then sets up some variables needed, checks to ensure Cryptor has been initialized, and calls either the CRYUtl_DecodeString or CRYUtl_EncodeString functions, depending on what it's supposed to do. If it succeeded, it returns the encrypted or decrypted string; otherwise, it sets the cErrorMessage property to an appropriate value and returns .NULL.

```
lparameters tcString, ;
  tcPassword, ;
  tnMethod, ;
  tcProcess
local lcPassword, ;
  lnMethod, ;
  lnLen, ;
  lcString, ;
  lnStatus
with This

* Ensure a string was passed.

  assert not empty(evl(tcString, '')) ;
    message 'Must pass string'

* Use the cPassword and nEncryptionMethod properties if
* the parameters weren't passed. Create other variables
* we need.

  lcPassword = iif(empty(evl(tcPassword, '')), ;
    .cPassword, tcPassword)
  lnMethod   = iif(vartype(tnMethod) <> 'N', ;
    .nEncryptionMethod, tnMethod)
  lnLen      = len(tcString)
  lcString   = space(lnLen)

* If Cryptor is initialized, try to process the string.

  if .InitializeCryptor()
    if tcProcess = 'decrypt'
      lnStatus = CRYUtl_DecodeString(tcString, ;
        @lcString, lnLen, lcPassword, lnMethod)
    else
      lnStatus = CRYUtl_EncodeString(tcString, ;
        @lcString, lnLen, lcPassword, lnMethod)
    endif tcProcess = 'decrypt'
    if lnStatus <> CRYPTOR_ERR_SUCCESS
      .cErrorMessage = 'Could not ' + tcProcess + ;
        ' string: ' + CRYUtl_GetErrorMessage(lnStatus)
      lcString = .NULL.
    endif lnStatus = CRYPTOR_ERR_SUCCESS
  else
    lcString = .NULL.
  endif .InitializeCryptor()
endwith
return lcString
```

The InitializeCryptor method is called from all methods in SFCryptor that use Cryptor functions. This method performs the function declarations and initialization. There are two advantages in doing these tasks this way rather than in the Init method of the class: the tasks are only performed if actually required and some properties that tell SFCryptor how to work with Cryptor (cCryptorPath, which specifies the location of XICRCORE.DLL, and nLoadMode, which specifies the load mode to use) can be set programmatically after instantiating the class.

```
local lcDLL, ;
  llReturn
with This
```

```
* If we're not already initialized, see if we can find
* XICrCore.dll.

  if not .lInitialized
    lcDLL    = fullpath('XICrCore.dll', ;
      addbs(.cCryptorPath))
    llReturn = file(lcDLL)
    if llReturn

* Declare the functions that we'll use in XICrCore.

      declare integer CRYIni_Initialize in (lcDLL) ;
        integer LoadMode
*** other function declarations omitted for brevity

* Initialize Cryptor.

      lnStatus = CRYIni_Initialize(.nLoadMode)
      if inlist(lnStatus, CRYPTOR_ERR_SUCCESS, ;
        CRYPTOR_ERR_ALREADY_INITIALIZED)
        .lInitialized = .T.
      else
        .cErrorMessage = 'Could not initialize ' + ;
          'Cryptor: status code ' + transform(lnStatus)
        llReturn = .F.
      endif inlist(lnStatus ...
    else
      .cErrorMessage = 'XICrCore.dll was not found.'
    endif llReturn
  else
    llReturn = .T.
  endif not .lInitialized
endwith
return llReturn
```

There are also methods to encrypt and decrypt files (Encrypt and Decrypt) and register and unregister files (Register and Unregister); all of these methods simply call the protected ProcessFile method to do the actual work. Feel free to look at the code for these methods yourself.

The ReleaseMembers method, called when the SFCryptor object is destroyed, cleans up by unregistering all registered files, uninitializing Cryptor, and clearing all Cryptor functions.

```
local lcFile, ;
  lnFlags, ;
  lnMethod, ;
  lnCount, ;
  lnFiles, ;
  lnStatus, ;
  laFiles[1], ;
  lnI
if This.lInitialized

* Unregister all files. We have to gather a list of files
* first, then unregister them, because you can't
* unregister a file while a list operation is in
* progress.

  lcFile   = space(260)
  lnFlags  = 0
  lnMethod = 0
  lnCount  = 0
  lnFiles  = 0
  lnStatus = CRYMan_List(CRYPTOR_REGLIST_FIRST, ;
    @lcFile, @lnFlags, @lnMethod, @lnCount)
  do while inlist(lnStatus, CRYPTOR_ERR_SUCCESS, ;
    CRYPTOR_ERR_END_OF_LIST)
    lnFiles = lnFiles + 1
    dimension laFiles[lnFiles]
    laFiles[lnFiles] = lcFile
    lnStatus = CRYMan_List(CRYPTOR_REGLIST_NEXT, ;
      @lcFile, @lnFlags, @lnMethod, @lnCount)
  enddo while inlist(lnStatus ...
  for lnI = 1 to lnFiles
    lnStatus = CRYMan_Unregister(laFiles[lnI])
  next lnI

* Uninitialize Cryptor.

  CRYIni_UnInitialize()

* Clear all functions.

  clear dlls CRYIni_Initialize, CRYIni_UnInitialize, ;
    CRYUtl_EncodeString, CRYUtl_DecodeString, ;
    CRYUtl_GetErrorMessage, CRYUtl_Encode, ;
    CRYUtl_Decode, CRYMan_Register, CRYMan_Unregister
endif This.lInitialized

* Carry on with the usual behavior.

dodefault()
```

Pricing and licensing

Cryptor 4.0 is $299, which provides 50 runtime licenses. An additional 500 runtime licenses can be purchased for $199. An unlimited license is $499. As of this writing, you can't order Cryptor directly from the Xitech web site. You can either email them a request to order Cryptor, or order it from one of their resellers, such as Hallogram (www.hallogram.com).

Since I started working with Cryptor, version 5.0 was released. I haven't looked at it, but the Xitech web site has details on the new features. Most attractive is a COM interface, eliminating the need to declare functions in a DLL. No pricing or licensing information is available on the Xitech web site; they really need to do some improvements there. Cryptor 4.0 is still available, so my guess is that version 5.0 is more expensive.

Summary

Cryptor is a great solution for the problem of preventing unauthorized access to VFP data, and can also encrypt strings (such as passwords stored in an INI file or the Windows Registry). I've used it for nearly a year with great success.

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, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", 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 Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com