Encryption the Fast (and Cheap!) Way

Doug Hennig

In the previous issue, Doug discussed a free library generously provided by Craig Boyd to compress and decompress files using the ubiquitous ZIP format. This time, he examines another free library from Craig, one that encrypts and decrypts strings and files.

Many VFP applications need to encrypt and decrypt data. Database connection strings, passwords, and sensitive data such as credit card or banking information should be encrypted rather than stored in clear text so prying eyes can't use the information.

There are numerous encryption libraries available. In fact, VFP even comes with one: the Crypto API example in the Solution Samples. However, many VFP developers have had problems with that example, the biggest one being that it gives different results on different operating systems. Once again, Craig Boyd to the rescue! He has created a library that's free, easy to use, and small (only 148K).

Here's a list of blog entries Craig has posted about his VFPEncryption library. You can read them all if you wish or jump to the last one for the most current download and documentation.

- http://tinyurl.com/357uah2
- http://tinyurl.com/338yhwa
- http://tinyurl.com/38plh6q
- http://tinyurl.com/383apg9
- http://tinyurl.com/36xqfae
- http://tinyurl.com/2wwooxo
- http://tinyurl.com/3xh6wtj
- http://tinyurl.com/36ref27
- http://tinyurl.com/3ypg8ba
- http://tinyurl.com/3645n9o
- http://tinyurl.com/365npby
- http://tinyurl.com/36amthj
- http://tinyurl.com/2vb9hlf

There are two versions of VFPEncryption. VFPEncryption71.FLL requires the Visual C++ 7.1 runtime, the same one used by VFP 9 (MSVCR71.DLL). VFPEncryption.FLL (that is, without the "71") requires the Visual C++ 9.0 runtime (MSVCR90.DLL). The two FLLs have the same functionality, so you'll likely want to use VFPEncryption71.FLL. If you use the wrong FLL for the Visual C++ runtime you have, you'll get a "FLL is invalid" error.

If you check out the documentation at http://tinyurl.com/2vb9hlf, you'll find there are five different types of encryption and four different modes. In addition, there's five different ways to pad the string to encrypt. How do you choose among all of these?

The easiest decision to make is if you have to follow a certain standard, such as that imposed by company policy or by a third-party (customer or vendor). However, if you're free to make your own choice, then go with the defaults VFPEncryption provides.

Cryptography is a complex subject, so I recommend reading Craig's blog posts and articles he links to if you're interested in background or more details.

The download for this article includes sample code showing how VFPEncryption works. DO TestVFPEncryption.PRG to start the sample. It displays the form shown in **Figure 1**. Select a sample from the list, click Show Code to see the code, then Run to run the example.

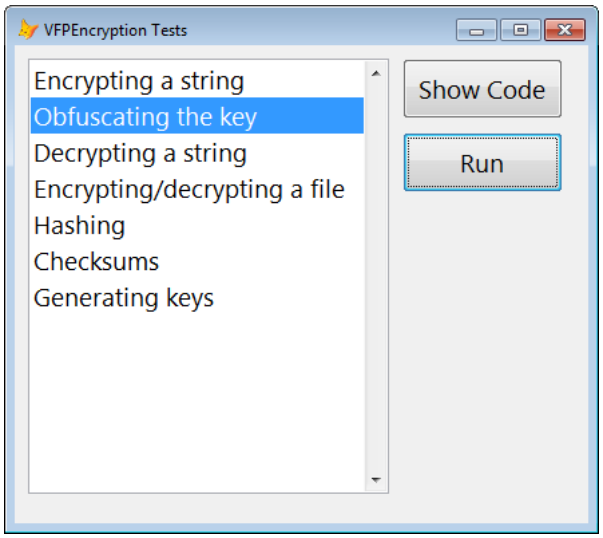

Figure 1. The sample form for this article makes it easy to see how VFPEncryption works.

Encrypting and decrypting a string

To encrypt a string, call Encrypt. It accepts the following parameters and returns the encrypted string:

- cStringToEncrypt: the string to encrypt.
- cSecretKey: the key to use for encryption. It has certain requirements for most types of encryption; see the next parameter. If you specify a key that's too short (such as 24 characters when 32 are required), VFPEncryption appends the key to itself to the desired number of characters.
- nEncryptionType: the encryption type, which is one of the following (optional; the default is 2):

  0 = Rijndael\AES 128 (requires a 16 byte key)
  1 = Rijndael\AES 192 (24 byte key)
  2 = Rijndael\AES 256 (32 byte key)
  4 = Blowfish (key between 1 and 56 bytes)
  8 = TEA (16 byte key)
  1024 = RC4 (key can be any length)

- nEncryptionMode: there are three different modes available (optional; the default is 0):

  0 = Electronic Code Book (ECB)
  1 = Cipher Block Chaining (CBC)
  2 = Cipher Feedback Block (CFB)
  3 = Output Feedback Block (OFB)

  This parameter does not apply to RC4 encryption (nEncryptionType = 1024).

- nPaddingMode: for block ciphers (nEncryptionMode = 1, 2, or 3), cStringToEncrypt is padded to a multiple of the block size (see the nBlockSize parameter). This parameter allows you to specify how this padding is done (optional; the default is 0):

  0 = Zeroes (NULLs)
  1 = Spaces (blanks)
  2 = PKCS7
  3 = ANSI X.923
  4 = ISO 10126

- nKeySize: the size of cSecretKey in bytes (optional).
- nBlockSize: the block size to pad cStringToEncrypt to (optional).
- cIV: the Initialization Vector (IV) used for block ciphers (nEncryptionMode = 1, 2, or 3). cIV is optional; if passed, it should match the specified nBlockSize in length.

The last four parameters are provided for compatibility with .Net and other encryption systems so strings you encrypt in VFP can be decrypted in those other systems and vice versa.

The key should be kept secret since it's the key (pun intended) to decrypting the string correctly. If someone determines your key, it's very easy for them to decrypt the string to retrieve the original information. You might not want to hard-code the key in your applications because someone decompiling it can easily discover the value. You might instead want to use some obfuscated code to return the key; the second sample ("Obfuscating the key") uses code written by Christof Wollenhaupt to do it.

I like to use RC4 (nEncryptionType = 1024) because you have more flexibility with the key. Also, it returns a string that's the same length as the original text, which is handy if you want to encrypt the contents of a field using a REPLACE statement.

Example:

```
lcOriginal = 'Craig Boyd rocks!'

* Use RC4.

lcRC4Key = 'Th1s sh0u1d be hard t0 guess'
lcRC4Encrypted = Encrypt(lcOriginal, ;
    lcRC4Key, 1024)

* The default AES 256.

lcAES256Key = 'Th1s sh0u1d be hard ' + ;
    't0 guess9#$%'
lcAES256Encrypted = Encrypt(lcOriginal, ;
    lcAES256Key)

* Blowfish.

lcBlowfishEncrypted = Encrypt(lcOriginal, ;
    lcRC4Key, 4)

* AES 256 with OFB.

lcAES256OFBEncrypted = Encrypt(lcOriginal, ;
```

```
    lcAES256Key, 2, 3)
```

To decrypt an encrypted string, call Decrypt. It accepts the same parameters as Encrypt and returns the decrypted string (assuming the key is the same as the one used to encrypt it).

Example:

```
lcDecrypted1 = Decrypt(lcRC4Encrypted, ;
    lcRC4Key, 1024)
lcDecrypted2 = Decrypt(lcAES256Encrypted, ;
    lcAES256Key)
lcDecrypted3 = Decrypt(lcBlowfishEncrypted, ;
    lcRC4Key, 4)
lcDecrypted4 = Decrypt(lcAES256OFBEncrypted, ;
    lcAES256Key, 2, 3)
```

Generating keys

If you need help coming up with a key to use for encryption functions, you can call GenerateKey to obtain a random key. It has the following parameters (all but the first are optional):

- nKeySize: the size of the key to be returned in bytes.
- lIncludeNumbers: .T. to include digits (0-9) in the key or .F. to exclude them.
- lIncludeUpper: .T. to include uppercase characters in the key or .F. to exclude them.
- lIncludeSpecial: .T. to include punctuation characters ({}|\\]?[\":;'><,./~!@#$%^&*()_+`-=) in the key or .F. to exclude them.

Encrypting and decrypting a file

To encrypt a file, call EncryptFile(cFileToEncrypt, cDestinationFile, cSecretKey [, nEncryptionType [, nEncryptionMode [, nPaddingMode [, nKeySize [, nBlockSize [, cIV]]]]]]). The first parameter is the name and path of the file to encrypt, the second is the name and path of the file to write the encrypted contents to (which cannot be the same as the original file), and the rest of the parameters are the same as those for Encrypt.

Similarly, to decrypt a file, call DecryptFile, which accepts the same parameters as EncryptFile. One interesting thing I found with DecryptFile is that unlike other functions, it gives an error if the key isn't the proper length rather than simply adjusting the key as necessary.

Note that unlike some commercial products such as Cryptor, VFPEncryption doesn't decrypt tables in memory and leave the copy on disk encrypted. That means if you want to use VFPEncryption as a cheap alternative to Cryptor, you have to decrypt the table to a temporary copy, open that table, then delete it once you're done with it.

Example:

```
strtofile('Craig Boyd rocks!', 'original.txt')
lcKey = 'Th1s sh0u1d be hard t0 guess9#$%'
EncryptFile('original.txt', 'encrypt.txt', ;
    lcKey)
DecryptFile('encrypt.txt', 'decrypt.txt', ;
    lcKey)
```

Hashing

Hashing is a specialized form of encryption: it uses a one-way algorithm to generate the encrypted value. "One-way" means you can't regenerate the original value from the hash value. Why would you want to do that? The typical case is handling passwords for user login.

One common design is to store user name and password combinations in a table. When the user logs in, find the record for the specified user name in the table and compare the stored password to the entered one. As Christof discusses in an article (http://tinyurl.com/2a5yju3), there are a few problems with this approach, basically coming down to preventing an unauthorized user from decrypting the password and therefore being able to log in. The solution is to not store the password at all but instead store the hash of the password.

Here's how it works:

- When the administrator assigns a password to a user, determine the hash value and store that instead of the password.
- When the user logs in, determine the hash value for the entered password.
- Compare the two hash values: if they don't match, the user entered the wrong password.

Because it's not possible to retrieve the original password from the stored value, there's no possibility of an unauthorized user obtaining the password from the table.

Craig discusses another use for hashing in his blog post at http://tinyurl.com/383apg9. He provides a utility (**Figure 2**) for generating license keys you can use in your applications to ensure the user has a valid license to use the application. One way to use this is to use something specific about the user's system, such as the hard drive serial number, as the key. If the user installs the application on a different system, the hash of the key won't be the same so the license won't be valid.

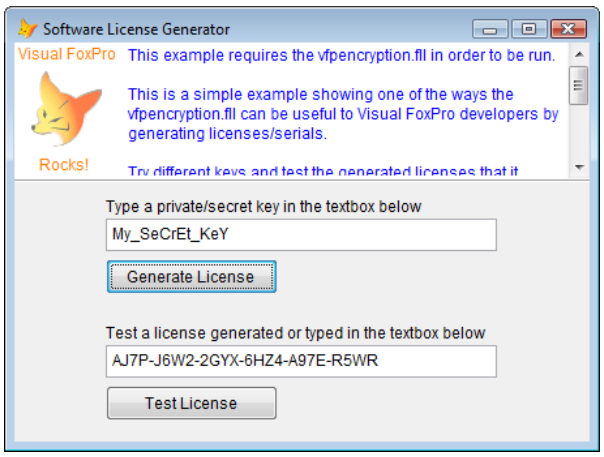

Figure 2. Craig's license generator can generate license keys for your applications.

VFPEncryption provides Hash and HashFile functions. Both accept the text to hash (a string in the case of Hash and the name and path of a file for HashFile) and an optional hash type value (4 is the default):

1 = SHA1 (a.k.a SHA160)
2 = SHA256
3 = SHA384
4 = SHA512
5 = MD5
6 = RIPEMD128
7 = RIPEMD160

To hash the contents of the fields in the current record in a table, call HashRecord(cAlias, [nHashType [, lIncludeMemos]]). Pass .T. for the last parameter to include the contents of memo fields.

Example:

```
* Hash a string.

lcOriginal   = 'Craig Boyd rocks!'
lcHashSHA512 = Hash(lcOriginal)
lcHashMD5    = Hash(lcOriginal, 5)

* Hash a file.

strtofile(lcOriginal, 'hashme.txt')
lcHashFile = HashFile('hashme.txt')

* Hash a record.

create cursor Test (Field1 C(10), Field2 I, ;
    Field3 C(60))
insert into Test values ('Record A', 1, ;
    'This is the first record')
lcHashRecord = HashRecord(alias())
```

Hash values are often displayed as HexBinary, so you can use STRCONV(HashValue, 15) to convert to that format.

A variant of a hash is a Hash-based Message Authentication Code or HMAC. This is a hash of a string in combination with a secret key. This can be used to simultaneously verify both data integrity (was the string altered?) and authenticity (did the string come from who we thought it came from?). VFPEncryption has a function to generate HMAC values: HMAC(cString, cKey [, nHashType]).

Example:

```
lcOriginal = 'Craig Boyd rocks!'
lcKey      = 'Th1s sh0u1d be hard t0 guess'
lcHMAC     = HMAC(lcOriginal, lcKey)
```

Checksums

Checksums are useful for determining if data was changed. Store the data and the checksum for the data. Later, calculate the checksum for the data and compare it to the stored checksum; if they aren't the same, the data was changed. You can also use a checksum as a short, simple hash of a value.

VFPEncryption includes three functions that calculate checksums: CRC(cString [, nCRCType]), CRCFile(cFile [, nCRCType]), and CRCRecord(cAlias [, nCRCType [, lIncludeMemos]]). In all three cases, specify 1 for nCRCType for a 16-bit checksum and 2 for a 32-bit value. Pass .T. for the last parameter of CRCRecord to include the contents of memo fields. Unlike the related VFP functions (SYS(2007) and SYS(2017)), which return strings, these functions return numeric values. Note that the VFP and VFPEncryption functions return different values for the same strings, so obviously they're using different algorithms.

Example:

```
* Calculate the checksum for a string.

lcOriginal = 'Craig Boyd rocks!'
lnCRC16    = CRC(lcOriginal, 1)
lnCRC32    = CRC(lcOriginal, 2)
lcCRC16VFP = sys(2007, lcOriginal)
lcCRC32VFP = sys(2007, lcOriginal, 0, 1)

* Calculate the checksum for a file.

strtofile(lcOriginal, 'crcme.txt')
lnCRCFile = CRCFile('crcme.txt')

* Calculate the checksum for a record.

lnSelect = select()
select 0
create cursor Test (Field1 C(10), Field2 I, ;
    Field3 C(60))
insert into Test values ('Record A', 1, ;
    'This is the first record')
lnCRCRecord = CRCRecord(alias())
```

Summary

Craig Boyd has created a very easy-to-use and inexpensive (free!) library we can add to our VFP applications to encrypt and decrypt strings and files. I've been using it for a couple of years with

great success in my applications and am sure you'll love it too.

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).