The Fox Hunt

Handling Passwords - I

By Whil Hentzen

Last month, the FoxPro Developer's Journal discussed a technique for masking password input through the use of echoing asterisks with every keystroke. Once you've implemented this (or another) technique, you'll run into several more issues. Let's discuss some of these.

First, you'll be storing these passwords in your user file, right? And since your user file is a DBF, it would be a pretty easy matter for anyone to read that DBF, either with FoxPro, or by importing the file from another piece of software like Excel. Obviously, we don't want to store the actual password in the user table - we should encrypt it.

At this point, most programmer's eyes start to sparkle. "This is the fun stuff - figuring out a clever algorithm to encrypt a string into another one." But let's not reinvent the wheel, and let's make sure we cover the fundamental principles involved in encryption. A reasonably secure algorithm will prevent users from determining the algorithm - even when they have examples of unencrypted and encrypted versions. In other words, you want to discourage reverse engineering.

If you use a simple substitution algorithm - where "A" is replaced by "B", "B" is replaced by "C", and so on - it is relatively easy to crack the algorithm by comparing several passwords with their encrypted versions:

```
Password    Encrypted
ABCDE       BCDEF
ANCHOR      BODIPS
HELLO       IFMMP
DOLLAR      EPMMBS
```

Here are three techniques that can help prevent the reverse engineering of your algorithm. Each of these will be discussed using a substitution algorithm as an example. First, instead of using the same offset, change it for each position in the password. For instance, bump the offset by 1 in the first position, by 2 in the second position, by 3 in the third, and so on.

```
Password    Encrypted
AAAAA       BCDEF
ANCHOR      BPFKUX
HELLO       IGNOT
DOLLAR      EQNOFX
```

Note that it is significantly more difficult to determine a pattern, because the same letter gets translated differently according to position. But still, a clever user could figure out the pattern - by noticing that the first letter "A" was a "B" in both AAAAA and ANCHOR and that the "R" in ANCHOR and DOLLAR becomes an "X."

So the next technique involves changing the offset according to some "random" pattern. One method is to use a series of strings that are added to the password, and to choose one of those strings via some difficult to discern method. For example, in the previous example, we incremented every string by "123456789." Here, we'd increment one string by "432093222," a second string by "992837412" and a third string by "002218127." Better yet, use character strings such as Rd5Kpt99E, adding the ASCII value of the character.

The trouble with this mechanism is that it starts to become a maintenance problem - you, the programmer, still have to keep track of these strings. An easy way to provide some variance without making it overly complicated is to use the first ten months of the year as ten character strings, and then to use one of those based on the seconds value of the system time. The following function demonstrates how this might be done.

```
************************************************************
function f_encrypt
************************************************************
*    Description: function to encrypt and unencrypt strings
*               :
*         Syntax: m.cResult = f_encrypt(<c String>, <n Method>)
*               :
*     Parameters: <c String> the string to encrypte
*               : <n Method> 1 to encrypt <c String>
*               :            2 to unencrypt <c String>
```

```
*               :
* Calling Sample: m.cEncrPW = f_encrypt("DOLLAR",1)
*               : m.cUnEncrPW = f_encrypt(m.cEncrPW,2)
*               :
*          Notes: unencrypted strings can be a maximum of 10 chars
*               : encrypted strings can be a maximum of 11 chars
*
para m.cString, m.nMethod
m.cResult = m.cString
*
* this is a conversion table against which we match up the
* string - which element we use depends on the current
* value of SECO()
*
* this provides a random translation key - passing the same
* string to this function only has a 1 in 10 chance of
* creating the same encrypted string
*
* the check digit on the end of the encrypted string tells
* us which element was used, as well as confusing the casual
* browse of the passwords, since the encrypted password
* isn't even the same length as the original password
*
declare aConvStr[10]
aConvStr[1]  = "JANUARYJAN"
aConvStr[2]  = "FEBRUARYFE"
aConvStr[3]  = "MARCHMARCH"
aConvStr[4]  = "APRILAPRIL"
aConvStr[5]  = "MAYMAYMAYM"
aConvStr[6]  = "JUNEJUNEJU"
aConvStr[7]  = "JULYJULYJU"
aConvStr[8]  = "AUGUSTAUGU"
aConvStr[9]  = "SEPTEMBERS"
aConvStr[10] = "OCTOBEROCT"
*
* handle all the goofy exceptions
*
do case
case para() < 2
 wait wind "Not enough parameters passed to f_encrypt"
case type("m.nMethod") <> "N"
 wait wind "Second parm (Method) must be numeric: 1 to encr, 2 to decr"
case m.nMethod = 1 and len(allt(m.cString)) > 10
 wait wind "First parm (String) during encryption must be maximum of 10 chars"
case m.nMethod = 2 and len(allt(m.cString)) > 11
 wait wind "First parm (String) during decryption must be maximum of 11 chars"
case ! inlist(m.nMethod, 1, 2)
 wait wind "Invalid method passed to zEncrypt"
case m.nMethod = 1
 *
 * encrypt the string
 * determine which of the ten strings to use by grabbing the units
 * part of the system time (a value from 0 to 9)
 *
 m.nTimeStr = val(right(time(),1))
 *
 * this grabs a string from January -> October
 *
 m.cConvStr = aConvStr[ m.nTimeStr + 1 ]
 *
 * we are adding the ASCII value of the Convert string to the
 * passed string, and then figuring out what the "chr" of that is
 *
 * loop through each position of the passed string
 *
 for i = 1 to len(allt(m.cString))
  m.cYY = chr( asc(subs(m.cString,i,1)) + asc(subs(m.cConvStr,i,1)) )
  m.cResult = stuff(m.cResult, i, 1, m.cYY )
```

```
 endfor
 *
 * encrypt the "check value" as well
 *
 m.cResult = m.cResult + chr( 128 + m.nTimeStr)
case m.nMethod = 2
 *
 * decrypt the string - using the reverse process
 *
 m.cConvStr = aConvStr[ asc(right(m.cString,1)) - 127 ]
 m.cString = left(m.cString, len(m.cString)-1 )
 m.cResult = m.cString
 for i = 1 to len(allt(m.cString))
  m.cYY = chr( asc(subs(m.cString,i,1)) - asc(subs(m.cConvStr,i,1)) )
  m.cResult = stuff(m.cResult, i, 1, m.cYY )
 endfor
othe
 wait wind "DEV: you passed a bad method to ZENCRYPT"
endc
return m.cResult
```

Note that we're using the same function to go both ways. This aids in maintenance - instead of having to keep track of two functions, we've just one more in our library. Keeping all the code for this function in one place will also help if we need to revise the function at some point - a change or enhancement made in one direction can immediately be incorporated in the other as well.

Once we've developed a mechanism for encrypting the password, however, we've still several other issues to consider. Next month, we'll finish up this mini-series by discussing how to prevent the use of common passwords and the creation of default passwords for new users.