Handy Handlers #5: Pack/Unpack

"Why should you care if you have nothing to hide?"
- J. Edgar Hoover

After a long break for the holidays and the Revolution Seminar at MacWorld, I'm finally back at my desk. Time away can bring a fresh perspective: at the end of Handy Handlers #4 I suggested the next column should describe a functional "if", but it occurs to me now that it's not really all that handy. So instead I'm starting the new year with something much more fun: cryptography.

During MacWorld, Runtime Revolution Ltd. announced a subset of features for the upcoming version 2.5, including "industrial-strength encryption" and SSL, so for the real thing you'll have to be patient just a bit longer. But in the meantime you can still have some fun using the software equivalent of a secret decoder ring, a pair of functions I call "Pack" and "Unpack".

Overview of Pack and Unpack

The Pack function returns the data passed to it in a compressed form, using base64 encoding on the output for easy storage and transmission as plain text. Optionally you can provide a password and the data will be encrypted as well. Unpack reverses the procedure, taking the compressed data and a password matching the one it was encrypted with, and returning that data in
its original form.

So for Pack, the basic steps we want to take are:

1. Get the data and a password.
2. Compress the data.
3. Encrypt it using the password.
4. Run it through the Base64 function
5. Return it.

To reverse this, Unpack needs to:

1. Take the encrypted data and a password
2. Convert it from base64 to binary
3. Decrypt it using the password.
4. Decompress it.
5. Return it.

Most of that's pretty simple, as you'll see. The tough part is Step 3: Just how do we encrypt the data?

Cryptography is not a trivial task if it is to provide real security. Experts devote years to crafting uncrackable algorithms, and other experts devote years to cracking them. Revolution user Mark Brownell has written a Blowfish implementation in LiveCode which uses an algorith so secure that it must be arbitrariliy limited to avoid violating US export laws.
I'll leave the serious cryptography for those who know what they're doing, and with the announcement about Revolution 2.5 hopefully it won't be long until we have very strong encryption built efficiently into the engine. But for now we can explore a simple algorithm I call MDX.
MDX stands for "MD5 plus Xor". It uses two popular techniques together to provide a very modest level of security more useful than using either alone. It's not the sort of thing you'd want to use for encrypting medical records or nuclear secrets, but it's definitely a step up from using plain Base 64. Let's see how it works:

MD5

MD5digest is a built-in LiveCode function the returns a 16-byte chunk of data which represents a sort of signature of whatever data was passed to it. You simply pass any data to the md5diget function and you get a 16 byte string. The string itself is meaningless; what's useful is that it's considered mathematically improbable for two different input strings to produce the same result. In essence, the result works like a fingerprint for the data, assuring with a high degree of confidence that any two strings producing the same md5digest result are identical.
Moreover, it's considered next to impossible to derive the source data just by examining this 16-bit fingerprint. This makes it ideal for storing passwords, since you can keep the data in a form that prevents anyone from actually seeing the password itself.
To decrypt the data with our Unpack function we'll need to be able to compare the password being used for that with the one that was used to encrupt it. By using the md5digest result of the password we never have to store the actual password itself.

So far so good, we can hide the password. But how do we encrypt the data? One way is to hide the password really well, by using that md5 result as the key for altering the data using Xor.

Xor

Xor is an operation that can be done on binary data to transform a set of bits to their opposite values. This implies of course that applying Xor to that data a second time returns them to their original state.
You can see Xor in action graphically in LiveCode: place an object on a card and put another one on top of it. In the Inspector, set the ink property of the topmost object to "srcXor" -- the control beneath it is then drawn with reversed colors. If you add another object and set its ink to "srcXor" also you'll see that where the two Xor'd objects overlap everything appears normal, since the reversing effect of the first Xor'd object is reversed again by the second.
Just as the srcXor ink reverses drawing, the bitXor operator does the same with binary data: if you Xor one value with another and then Xor that result the same way, you get what you started with.

Putting It Together

Keep in mind that while using LiveCode's compress function makes data unreadable to humans, it uses the open GZip standard, so anyone with a GZip decompression tool could read it in one step. Same with the base64encode function we use to convert the compressed data to plain text.
To prevent this we need to alter the data so that any attempt to decompress it with normal GZip will fail, and we need to do so in an orderly fashion so we can restore the data again later.
To alter the data we'll step through it character by character (byte by byte), applying an Xor operation to it. And we'll get the value to use in that Xor operation from our MD5-derived string, getting the next character in the MD5 string each time through the loop.
So the heart of our encryption looks like this, taking data passed in pData and returning the encrypted form in tCryptoText:

```
-- Get MD5 digest:
put md5digest(pPassword) into tKeyString
put len(tKeyString) into tKeyStringLen
--
put compress(pData) into pData
--
-- Apply it with Xor to the data one byte at a time:
put 0 into i
put empty into tCryptoText
repeat for each char k in pData
  add 1 to i
```

```
  if i > tKeyStringLen then put 1 into i
  put char i of tKeyString into tKeyChar
  put numtochar( chartonum(k) bitxor
chartonum(tKeyChar))\
    after tCryptoText
end repeat
```

Sometimes we may want to use Pack and Unpack to simply compress and base64 data without encryption, and to accommodate this we can add a flag to the data used by Unpack to determine if the data needs to undergo that password transformation. We'll go one step further and make that flag a number so we can later modify these functions to add support for different encryption schemes as they become available (0 = no encryption, 1 = MDX, 2 could equal Blowfish, etc.). In fact we'll make it a two-digit number so we can support up to 99 different encryption methods if needed.

And lastly, since "Pack" and "Unpack" are such generic terms, we'll modify these name slightly to reduce the chance of naming conflict if you later use another library that has functions with the same names, or if functions with these names ever wind up in the engine itself. I tend to preface most of the handlers in my library with "fw", short for "Fourth World", so I can distinguish them from handlers borrowed from other libraries.

After adding a few lines to handle the encryption method flag along with the compress and base64 functions we get the full version of our Pack and Unpack functions:

```
-------------------------------------------------
----------
-- fwPack/fwUnpack
--
-- Generic functions for packing data for Internet
transmission,
-- with an optional modest level of password-protected
security.
--
-- In the simplest form, data is first compressed with
gzip and
-- then encoded with Base64 for robust storage or
transmission.
--
```

```
-- Data can be optionally protected by supplying a
password,
-- using a very lightweight but fast "mdx" (MD5+Xor)
algorithm.
-- DO NOT USE THIS ALGORITHM IF YOU REQUIRE STRONG
ENCRYPTION.
--
----------------------------------------------------
----------
--
-- fwPack
--
function fwPack pData, pPassword, pEncryptionMethod
  local tMd5, tRandSeed, tMax, tOffsetsA
  local tStackName, tSaveVis, tFile
  --
  if pPassword is empty then
    -- Simply mark the data as not password-protected
    -- and compress it without further modification:
    put "00"&compress(pData) into pData
  else
    -- Get MD5 digest:
    put md5digest(pPassword) into tKeyString
    put len(tKeyString) into tKeyStringLen
    --
    put compress(pData) into pData
    --
    -- Apply it with Xor to the data one byte at a
time:
    put 0 into i
    put empty into tCryptoText
    repeat for each char k in pData
      add 1 to i
      if i > tKeyStringLen then put 1 into i
      put char i of tKeyString into tKeyChar
      put numtochar( chartonum(k) bitxor
chartonum(tKeyChar))\
        after tCryptoText
```

```
    end repeat
    --
    -- Mark the data as password-protected:
    put "01" & tCryptoText into pData
  end if
  -- Convert to common low ASCII:
  return base64encode(pData)
end fwPack
--
--
-- fwUnpack
--
function fwUnPack pData, pPassword
  local tEncryptionMethod
  local tMd5, tRandSeed, tMax, tOffsetsA
  --
  -- Convert from base64 back to binary:
  put base64decode(pData) into pData
  -- Check and remove password-protection flag:
  put char 1 to 2 of pData into tEncryptionMethod
  delete char 1 to 2 of pData
  --
  switch tEncryptionMethod
  case "00" --no encryption
    break
    --
    case "01" -- mdx
      -- Get MD5 digest:
      put md5digest(pPassword) into tKeyString
      put len(tKeyString) into tKeyStringLen
      --
      -- Apply it with Xor to the data one byte at a
time:
      put 0 into i
      put empty into tClearText
      repeat for each char k in pData
        add 1 to i
        if i > tKeyStringLen then put 1 into i
```

```
        put char i of tKeyString into tKeyChar
        put numtochar( chartonum(k) bitxor
chartonum(tKeyChar))\
          after tClearText
      end repeat
      put tClearText into pData
      break
      --
  end switch
  --
  -- Attempt to decompress data, throwing an error if
not valid:
  try
    get decompress(pData)
    catch errNo
    throw "Wrong password"
    exit to top
  finally
    return it
  end try
end fwUnPack
```

About the Author

Richard Gaskin is Ambassador of Fourth World Systems, a Los Angeles-based consultancy specializing in multi-platform software development and LiveCode training. With 15 years' experience, Richard has delivered dozens of applications for small businesses and Fortune 500 companies on Mac OS, Windows, UNIX, and the World Wide Web.
http://www.fourthworld.com