Conversion Tools - Feature #3363

implement a replacement for the XCODE utility that can also handle decrypting source code

10/25/2017 11:34 AM - Greg Shah

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

History

#1 - 10/25/2017 11:58 AM - Greg Shah

- File xcode_encrypted_src_example_and_erlang_dx_code.zip added

In the 4GL, there is a utility called XCODE which can be used to encrypt source code in a manner that it can be decrypted given the right key. It does not provide a decryption mechanism. I think the Progress compiler does the decryption, assuming you set the right key.

The command line interface:

```
Usage: xcode [-d dir] [-k key] [-l] [-c] [filenames] [-]
 -d        directory for xcoded output files.
 -k        key to use in encryption.
 -l        convert filename to lowercase.
 -         take filenames from standard input.
 filenames files to encrypt.
```

If you do not specify a key, then it will use the default key "Progress". Yes, really.

The approach used by Progress is not really encryption, at least it is not using any valid crypto methods that would be accepted under the term "cryptography". I think the approach shares some similarities with the ENCODE approach (see #27).

I am attaching some open source code I found (written in Erlang) that purports to be able to encrypt and decrypt 4GL source code in a compatible way with Progress. It uses a 16-bit CRC approach inside a specific algorithm. The source code is licensed under the MIT license, so it is fair game. I think the best approach is to document the spec for the algorithm using the dx code and then write a from scratch implementation in Java. This is the same approach I took in #27.

One interesting thing that is immediately seen in the sample, is that the default key used is "Progress". I have proven that running XCODE with -k "Progress" will generate the same output as running XCODE without any -k option (with the so called default key).

This XCODE facility should not ever be used as a security mechanism. But that doesn't mean that it isn't in use. In order to deal with projects that have been encoded this way, we will create our own encode/decode program in Java. Then we can enhance the conversion process to read in "encrypted" files and process everything normally from there.

I don't know yet how to detect if something is encrypted or not, though I suspect there may be a signature at the end of the file.

Is it possible to decode encrypted program (XCODE utility)? (short answer: no tools are available)
Paid Service to Decrypt 4GL Source Code

Erlang Program to Encrypt/Decrypt 4GL Source Code (MIT licensed)
Concise Guide to Erlang
Erlang Documentation

Here is that code:

```
temporarily redacted
```

Based on this open source dx code, the task seems pretty straightforward.

#2 - 07/17/2018 08:57 AM - Greg Shah

- File xcode_encrypted_src_example.zip added

#3 - 07/17/2018 08:57 AM - Greg Shah

- File deleted (xcode_encrypted_src_example_and_erlang_dx_code.zip)

#4 - 07/17/2018 02:37 PM - Greg Shah

The XCODEd output is always 1 byte larger than the input. The byte is 0x13 and it is prefixed as the first byte in the XCODEd file. It might be a length of the encoded bytes. If so, other tests of XCODE should show this and the value is probably variable length too. It may just be a simple integrity check mechanism.

hello.p (18 bytes, ASCII encoded)

```
message "Hello!".
```

hello.p in hexadecimal notation:

```
0000000  6d 65 73 73 61 67 65 20 22 48 65 6c 6c 6f 21 22
0000010  2e 0a
```

encrypted_src/hello.p.encrypted_with_test (19 bytes, key is "test")

```
0000000  13 0b 7b d3 15 61 01 07 57 32 1b de 68 8c 37 72
0000010  2b e1 9b
```

encrypted_src/hello.p.encrypted_with_Progress (19 bytes, key is "Progress")

```
0000000  13 1b 60 76 44 5b 66 1c 2b 0a 60 4d d5 d5 57 80
0000010  83 15 27
```

encrypted_src/hello.p.encrypted_with_default_key (19 bytes, output is the same as using the key "Progress")

```
0000000  13 1b 60 76 44 5b 66 1c 2b 0a 60 4d d5 d5 57 80
0000010  83 15 27
```

We thus know the following:

- Dropping the fixed lead byte of 0x13 means that the encoded output has one encoded byte for every input byte, with no padding.
- The output is a function of the input key.
- The result is reversible (given the same key, the encoded text can be decoded into the original text).

The most obvious cipher approach to implement such output is the XOR Cipher. For this approach to be of any use, the byte being used for encoding ("adding" via xor to each input byte) must not be the same for the entire file. Otherwise decoding the input would be as simple as trying 256 possible byte values against the entire file and one of them would work.

This means that the encoding byte must change as the input is encoded. Using the above data:

- The first character of the input text is "m" which is 0x6d (binary 0110 1101).
- The first character of non-lead byte output using key "test" is 0x0b (binary 0000 1011).
- The xor value for the first character must be 0x66 (binary 0110 0110, lowercase "f" in ASCII).

- The second character of the input text is "e" which is 0x65 (binary 0110 0101).
- The second character of non-lead byte output using key "test" is 0x7b (binary 0111 1011).
- The xor value for the second character must be 0x1e (binary 0001 0110, which is a non-visual "record separator" character in ASCII).

This tells us that the simple approach of rotating through the key text is not used. In fact, the key text is not used without modification.

Knowing how ENCODE works, the CRC-16 would be expected to modify the key in a circular buffer and whatever byte was modified would be used as the xor value to be "added" to encode the next byte of the text.

Useful articles on CRC:

https://en.wikipedia.org/wiki/Cyclic_redundancy_check
http://www.qtcentre.org/threads/24881-CRC-16-0xA001-polynomial
https://stackoverflow.com/questions/23638939/crc-16-ibm-reverse-lookup-in-c
http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html

The next step will be to modify the ENCODE implementation to generate xor bytes based on the initial key text. ENCODE uses a 16 byte circular buffer that additively modifies the resulting hash based on the CRC-16-IBM (with reversed polynomial 0xA001) as it walks through the input text from left to right.

We can easily test if there is a limit to the size of the key text that is honored, which may tell us the size of the buffer used. This can be determined by changing key values at the end of longer keys (i.e. using two keys with the same prefix text but differing deep in the key).

Files

xcode_encrypted_src_example.zip	1.22 KB	07/17/2018	Greg Shah