Celare – An **RC4** compatible encryption algorithm

Introduction

Celare is a custom built encryption algorithm based strongly on the RC4 algorithm of RSA. It is a stream cipher algorithm using a cipher block size of 256 bytes and a key size of 256 bytes. If the key size is not exactly 256 bytes, it is repeated until it pads out to 256 bytes.

The RC4 algorithm is the very same algorithm used for current encryption systems such as WEP - used for wireless security. Due to the age of the algorithm, it's becoming less and less secure as computing power, and hence brute forcing power increases. In saying this however, it's adequately secure for small scale systems which consistently change their encryption keys.

History

RC4 was designed by Ron Rivest of RSA Security in 1987; while it is officially termed "Rivest Cipher 4", the RC acronym is alternatively understood to stand for "Ron's Code".

Celare is a custom built derivative of the RC4 algorithm. It differs in creating a new key-scheduling algorithm [KSA] as to provide a new level of secrecy in an already insecure encryption algorithm.

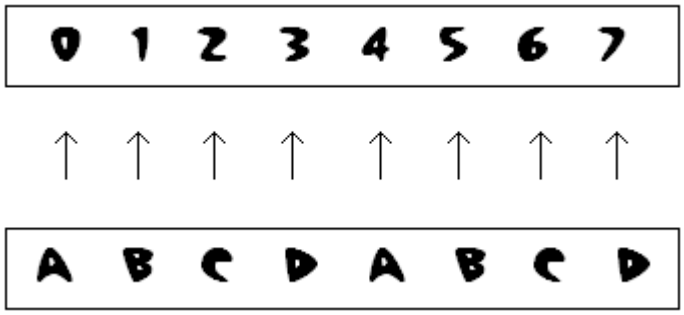
Intended Audience & Purpose

This document aims to provide a detailed explanation of how the *Celare* algorithm works using textual and visual aids, along with the C++ source code of an example implementation of the algorithm.

The Algorithm

Celare starts by creating two streams. The first stream is a series of numbers ranging from 0 to 255, 256 bytes in total. The second stream is the key, repeated to pad out 256 bytes of data – making both blocks of equal size.

The diagram below illustrates the idea with an 8 byte version of the algorithm and the key "abcd".



Essentially these two streams (after further ciphering) are combined to produce a *keystream* which will ultimately be combined with the plaintext input, via an exclusive or [XOR] operation - producing the cipher text. This combined cipher is known as the *Vernam Cipher*.

The decryption process is exactly the same due to the fact that the underlying operation of *Celare* is binary XOR and this operation is reversible. Of course, the same key must be used or the *Vernam Cipher* block will be different and actually produce a second level of encryption, rather than the expected decryption.

The 'Further Ciphering'

At this point we have two streams set up: one containing unsigned numbers ranging from 0 to block size – 1, and another containing the key (repeated if necessary to pad out the stream to the correct size).

Next comes the production of the *keystream*. This step is known as the pseduo-random generation algorithm [PRGA] and basically takes the first stream and re-orders the numbers, using the second stream as an integral part of the calculation for doing so.

This effect creates a cipher block that is different depending on the key supplied, which shows how the key plays the role of the instigator in the resulting cipher text. A different key will produce a different keystream, which consequetly will produce different cipher text.

The method the creation of the *keystream* is described below:

Assuming the first stream is named Box1 and the second stream is called Box2.

Box1 is traversed from the beginning to the end, i.e. From element 0 to block size – 1. Each interation will swap the current element with another element based off the second stream Box2, in the following manner:

x = (x + Box1[current iteration] + Box2[current iteration]) % block size;

Following this algorithm for the 8 byte example streams above, the following calculations are produced:

x = 0
current iteration = **0**

x = (0 + 0 + 97) % 8 = 1
swap(0,1)

x = 1
current iteration = **1**

x = (1 + 1 + 98) % 8 = 4
swap(1,4)

x = 4
current iteration = **2**

x = (4 + 2 + 99) % 8 = 1
swap(2,1)

x = 1
current iteration = **3**

x = (1 + 3 + 100) % 8 = 0
swap(3,0)

x = 0
current iteration = **4**

x = (0 + 0 + 97) % 8 = 1
swap(4,1)

x = 1
current iteration = **5**

x = (1 + 5 + 98) % 8 = 0
swap(5,0)

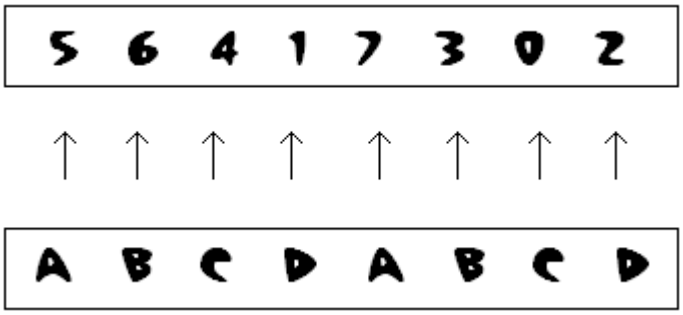
x = 0
current iteration = **6**

x = (0 + 6 + 99) % 8 = 1
swap(6,1)

x = 1
current iteration = **7**

x = (1 + 7 + 100) % 8 = 4
swap(7,4)

Ultimately Box1 is re-ordered based off the current element of Box1, plus the current element of Box2, plus the result of this previous operation (0 if this is first operation), finally modulo the block size to ensure the resulting number is in the correct range. This produces the following new *keystream*:



At this point of the algorithm, the second stream holding the key is no longer needed because it's work in re-ordering the first stream into a *keystream* is now complete. Essentially the only remaining operation is to combine the new cipher with the plain text, creating the cipher text.

The Key-scheduling Algorithm [KSA]

The KSA is the final algorithm responsible for combing the *keystream* with the plain text, producing cipher bytes. Each cipher byte output iteration also causes a new permutation of the cipher block – meaning that for every cipher byte created from the *keystream*, the *keystream* changes making it very difficult to decipher.

The KSA of the *Celare* algorithm operates in the following manner:

An index pointer *i* is maintained, initially pointing to the second element (0 based index of 1) and incremented each iteration to point to the next element of the *keystream*.

Another index pointer *j* is calculated by taking the byte at the position indicated by the *i* index, adding this to the value of the previous result of this operation (or 0 if this is the first iteration) and finally modulo the size of the cipher block to keep the *j* index within range.

Now there are two indexes, *i* and *j* both pointing to elements of the *keystream* cipher block. These two elements are then swapped. This is considered to be the single permutation executed each iteration of the key-scheduling algorithm.

After the swap has been completed, a new index *t* is computed which is the sum of the two values of the *keystream* pointed to by *i* and *j.* This value is then kept in range by using a modulo of the size of the cipher block.

Finally the plain text byte currently being pointed to is combined with the cipher byte of the *keystream,* indexed by *t*. The combination is a binary exclusive OR operation.

This process [KSA] is then repeated continuously until the entire input has been covered – each time taking two bytes from the *keystream* and swapping them, then selecting a cipher byte based off the last swap and XOR'ing it with the current plain text input byte; finally creating a full cipher text stream – completing the algorithm.

The Pseduo-random Generation Algorithm

```
1:    fill Box1 with incrementing numbers from 0
2:    fill Box2 with key (repeat if necessary)
3:    j = 0
4:    for i = 0 to Blocksize-1
5:    j = (j + Box1[i] + Box2[i]) % Blocksize
6:    swap Box1[i] and Box1[j]
```

The Key-scheduling Algorithm

```
1:    for x = 0 to PlainTextInputSize-1
2:    i = (i + 1) % Blocksize
3:    j = (j + Keystream[i]) % Blocksize
4:    swap Keystream[i] and Keystream[j]
5:    t = (Keystream[i] + Keystream[j]) % Blocksize
6:    PlainTextInput[x] = (PlainTextInput[x] XOR t)
```

```
/*-------------------------------------------------------------------------------
*
*       Implementation of the Celare encryption algorithm.
*
*-------------------------------------------------------------------------------*/
void Encrypt(char* pszInput, DWORD dwInputLength, char* pszKey, DWORD dwKeyLength)
{
        /*-----------------------------------------------------
        *       Internal encryption states and cipher blocks.
        *-----------------------------------------------------*/
        unsigned char Sbox[256], Sbox2[256];
        unsigned int i, j, t, x;

        /*-----------------------------------------------
        *       Counting variables for stream operations.
        *-----------------------------------------------*/
        unsigned char temp , k;
        i = j = k = t = x = 0;
        temp = 0;

        /*-------------------------------------------------
        *       Clear both streams to 0's before starting.
        *-------------------------------------------------*/
        ZeroMemory(Sbox, sizeof(Sbox));
        ZeroMemory(Sbox2, sizeof(Sbox2));

        /*---------------------------------------------------
        *       Fill first stream with incrementing numbers.
        *---------------------------------------------------*/
        for (i = 0; i < 256; i++)
        {
            Sbox[i] = i;
        }

        j = 0;

        /*-----------------------------------------------------------
        *       Copy key (repeating if necessary) into second stream.
        *-----------------------------------------------------------*/
        if (dwKeyLength)
        {
                for (i = 0; i < 256; i++)
                {
                        if (j == dwKeyLength)
                        {
                                j = 0;
                        }

                        Sbox2[i] = pszKey[j++];
                }
        }

        j = 0;

        /*-------------------------------------------------------
        *       The pseudo-random generation algorithm [PSGA].
        *-------------------------------------------------------*/
        for (i = 0; i < 256; i++)
        {
                /*-----------------------------------------------------------------------
                *       Acquire the secondary swapping index j.
                *-----------------------------------------------------------------------*/
                j = (j + (unsigned long) Sbox[i] + (unsigned long) Sbox2[i]) % 256;

                /*-------------------------------------------------
                *       Swap current iteration's index with j.
                *-------------------------------------------------*/
                temp = Sbox[i];
                Sbox[i] = Sbox[j];
                Sbox[j] =  temp;
        }

        i = j = 0;

        /*-----------------------------------------
        *       The key-scheduling algorithm [KSA].
        *-----------------------------------------*/
        for (x = 0; x < dwInputLength; x++)
        {
                /*-------------------------------------------------
                *       Acquire the permutation indexes.
                *-------------------------------------------------*/
                i = (i + 1) % 256;

                j = (j + (unsigned long) Sbox[i]) % 256;
```

```
                /*-----------------------------------------------
                *       Create permutation of keystream cipher.
                *-----------------------------------------------*/
                temp = Sbox[i];
                Sbox[i] = Sbox[j];
                Sbox[j] = temp;

                t = ((unsigned long) Sbox[i] + (unsigned long) Sbox[j]) % 256;

                k = Sbox[t];

                /*-----------------------------------------------
                *       Create the cipher text byte in input buffer.
                *-----------------------------------------------*/
                pszInput[x] = (pszInput[x] ^ k);
        }
}
```

Author: Martin Rue
Date: 24/02/2006