Block TEA (Tiny Encryption Algorithm)

Wheeler & Needham's Tiny Encryption Algorithm is a simple but powerful encryption algorithm (based on a 'Feistel cipher').

I've long been impressed by the combination of simplicity & effectiveness: while vulnerable to determined cryptanalysis such as related-key attack, TEA is a light-weight solution more appropriate for some applications than 'industrial strength' approaches such as AES.

This is a JavaScript implementation of the (corrected) 'Block TEA' or 'large block' version of the algorithm (also dubbed 'xxtea') with a wrapper to enable it to work on (Unicode) text strings.

This is a simple but highly effective DES-style encryption algorithm which can be useful for web applications which require security or encryption. It provides secure cryptographically strong encryption in a few lines of concise, clear (JavaScript) code.

Functional demo: enter password & plaintext (or previously-encrypted text)

Password encryptioñ-pw

Plaintext søme highly secret text to be encrypteđ

Encrypted text D9959BObdYPHiSEjLvyMSFUnDPxlv8vqyI169gA3YIznWUmg88r4s0QnoUQ= 0.100ms

Decrypted text document.queSelector is not a function

Wheeler & Needham say the Block TEA version is faster than the original (64-bit block version) when encrypting longer blocks (over 16 chars), and is more secure ('a single bit change will change about one half of the bits of the entire block, leaving no place where the changes start'). It is also simpler to implement for encrypting arbitrary-length texts (being variable block size, it requires no 'mode of operation'). For an implementation of the original algorithm, see tea.html.

TEA uses a 128-bit key, which could (for increased security) be an encrypted (or hashed) form of the supplied password. Here I simply convert the first 16 characters of the password into longs to generate the key. The password might be a user-supplied password, or an internal system password. A system password will be more secure if it avoids plain-text (e.g. 'dVr4t%G§Uu-mz7+8').

Wheeler & Needham's original formulation (in C) of corrected block TEA (aka xxtea) was as follows:

```
#define MX (z>>5^y<<2) + (y>>3^z<<4)^(sum^y) + (k[p&3^e]^z);

long btea(long* v, long n, long* k) {
  unsigned long z=v[n-1], y=v[0], sum=0, e, DELTA=0x9e3779b9;
  long m, p, q ;
  if (n > 1) {          /* Coding Part */
    q = 6+52/n ;
    while (q-- > 0) {
      sum += DELTA;
      e = sum >> 2&3 ;
      for (p=0; p<n-1; p++) y = v[p+1], z = v[p] += MX;
      y = v[0];
      z = v[n-1] += MX;
    }
    return 0 ;
```

```
  } else if (n < -1) {  /* Decoding Part */
    n = -n ;
    q = 6+52/n ;
    sum = q*DELTA ;
    while (sum != 0) {
      e = sum>>2 & 3;
      for (p=n-1; p>0; p--) z = v[p-1], y = v[p] -= MX;
      z = v[n-1];
      y = v[0] -= MX;
      sum -= DELTA;
    }
    return 0;
  }
  return 1;
}
```

I needed to encrypt text, not binary data, so I have built on this so that it operates on text rather than just on numeric arrays, and also rearranged it slightly so that *p* is not referenced outside the *for* loop (valid in C, but not always in other languages). The ciphertext is encoded as Base64 so that it can be safely stored and/or transmitted without troublesome control characters causing problems. The plaintext is first converted to UTF-8 so that the script is multi-byte-character safe.

If you want to use the encrypted text in a URL, the Base64 '+', '/', '=' characters can cause problems if not escaped, so you can convert the standard base64 string to base64url by

```
const b64url = b64.replace(/+/g, '-').replace(/\//g, '_').replace(/=/g, '.');
```

(and the reverse before decrypting the encrypted text, of course).

In other languages: remember ***always*** to use either ***unsigned right-shift*** *operators* or unsigned type declarations, according to features available in the language – signed right shift operations will fail; also, in strToLongs(), to avoid running off the end of the string, some languages may need the string to be padded to a multiple of 4 characters, with the equivalent of `for (var p=0; p<3-(s.length-1)%4; p++) s += '\0';`.

For an explanation of the operation of the TEA algorithm, and cryptography in general, an excellent book is *Information Security Intelligence: Cryptographic Principles & Applications* by Tom Calabrese (available from Amazon.com). There is also a good article explaining TEA operation and cryptanalysis by Matthew Russell from York University and a short article in Wikipedia.

Note: if you are interested in cryptanalysis of TEA, bear in mind that there are 4 versions described in 3 documents: the original *TEA*, then *Extensions to TEA* (addressing weaknesses in TEA and also describing Block TEA), and *Corrections to Block TEA* (aka xxtea). Block TEA is a variable-width block cipher with a number of benefits over the original. This page implements the last of these, xxtea.

If you really want industrial-strength encryption, I have also implemented a version of AES.

For some security applications, a cryptographic hash is more appropriate than encryption – if you are interested in a hash function, see my implementation of SHA-1.

See below for the source code of the JavaScript implementation, also available on GitHub.

This should be straightforward to translate into other languages if required. If needed, I have base64 conversion routines and UTF-8 conversion routines, and David Chambers has written a good-looking btoa()/atob() polyfill (also useful for targeting IE9-).

Note: this script was **revised** in **October 2009** to handle multi-byte UTF-8 character strings, and to encapsulate it

within a *Tea* namespace. It was revised in **June 2014** to use `btoa()` & `atob()` for base64 encoding, and `en/decodeURIComponent()` & `un/escape()` for UTF-8 encoding; the previous version is available for reference.

If you have any queries or find any problems, contact me at scripts-crypt@movable-type.co.uk.

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* Block TEA (xxtea) Tiny Encryption Algorithm                       (c) Chris Veness 2002-2019  */
/*                                                                                   MIT Licence  */
/* www.movable-type.co.uk/scripts/tea-block.html                                                  */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */


/**
 * Tiny Encryption Algorithm. David Wheeler & Roger Needham, Cambridge University Computer Lab.
 *
 * www.movable-type.co.uk/scripts/tea.pdf   - TEA, a Tiny Encryption Algorithm (1994)
 * www.movable-type.co.uk/scripts/xtea.pdf  - Tea extensions (1997)
 * www.movable-type.co.uk/scripts/xxtea.pdf - Correction to xtea (1998)
 */
class Tea {

    /**
     * Encrypts text using Corrected Block TEA (xxtea) algorithm.
     *
     * @param   {string} plaintext - String to be encrypted (multi-byte safe).
     * @param   {string} password - Password to be used for encryption (1st 16 chars).
     * @returns {string} Encrypted text (encoded as base64).
     */
    static encrypt(plaintext, password) {
        plaintext = String(plaintext);
        password = String(password);

        if (plaintext.length == 0) return ''; // nothing to encrypt

        //  v is n-word data vector; converted to array of longs from UTF-8 string
        const v = Tea.strToLongs(Tea.utf8Encode(plaintext));
        //  k is 4-word key; simply convert first 16 chars of password as key
        const k = Tea.strToLongs(Tea.utf8Encode(password).slice(0, 16));

        const cipher = Tea.encode(v, k);

        // convert array of longs to string
        const ciphertext = Tea.longsToStr(cipher);

        // convert binary string to base64 ascii for safe transport
        const cipherbase64 = Tea.base64Encode(ciphertext);

        return cipherbase64;
    }


    /**
     * Decrypts text using Corrected Block TEA (xxtea) algorithm.
     *
     * @param   {string} ciphertext - String to be decrypted.
     * @param   {string} password - Password to be used for decryption (1st 16 chars).
     * @returns {string} Decrypted text.
     * @throws  {Error}  Invalid ciphertext
     */
    static decrypt(ciphertext, password) {
        ciphertext = String(ciphertext);
        password = String(password);

        if (ciphertext.length == 0) return '';  // nothing to decrypt

        //  v is n-word data vector; converted to array of longs from base64 string
        const v = Tea.strToLongs(Tea.base64Decode(ciphertext));
        //  k is 4-word key; simply convert first 16 chars of password as key
        const k = Tea.strToLongs(Tea.utf8Encode(password).slice(0, 16));
```

```
        const plain = Tea.decode(v, k);

        const plaintext = Tea.longsToStr(plain);

        // strip trailing null chars resulting from filling 4-char blocks:
        const plainUnicode = Tea.utf8Decode(plaintext.replace(/\0+$/, ''));

        return plainUnicode;
    }


    /**
     * XXTEA: encodes array of unsigned 32-bit integers using 128-bit key.
     *
     * @param   {number[]} v - Data vector.
     * @param   {number[]} k - Key.
     * @returns {number[]} Encoded vector.
     */
    static encode(v, k) {
        if (v.length < 2) v[1] = 0;  // algorithm doesn't work for n<2 so fudge by adding a null
        const n = v.length;
        const delta = 0x9e3779b9;
        let q = Math.floor(6 + 52/n);

        let z = v[n-1], y = v[0];
        let mx, e, sum = 0;

        while (q-- > 0) {  // 6 + 52/n operations gives between 6 & 32 mixes on each word
            sum += delta;
            e = sum>>>2 & 3;
            for (let p = 0; p < n; p++) {
                y = v[(p+1)%n];
                mx = (z>>>5 ^ y<<2) + (y>>>3 ^ z<<4) ^ (sum^y) + (k[p&3 ^ e] ^ z);
                z = v[p] += mx;
            }
        }

        return v;
    }


    /**
     * XXTEA: decodes array of unsigned 32-bit integers using 128-bit key.
     *
     * @param   {number[]} v - Data vector.
     * @param   {number[]} k - Key.
     * @returns {number[]} Decoded vector.
     */
    static decode(v, k) {
        const n = v.length;
        const delta = 0x9e3779b9;
        const q = Math.floor(6 + 52/n);

        let z = v[n-1], y = v[0];
        let mx, e, sum = q*delta;

        while (sum != 0) {
            e = sum>>>2 & 3;
            for (let p = n-1; p >= 0; p--) {
                z = v[p>0 ? p-1 : n-1];
                mx = (z>>>5 ^ y<<2) + (y>>>3 ^ z<<4) ^ (sum^y) + (k[p&3 ^ e] ^ z);
                y = v[p] -= mx;
            }
            sum -= delta;
        }

        return v;
    }


    /**
     * Converts string to array of longs (each containing 4 chars).
     * @private
     */
    static strToLongs(s) {
        // note chars must be within ISO-8859-1 (Unicode code-point <= U+00FF) to fit 4/long
```

```
        const l = new Array(Math.ceil(s.length/4));
        for (let i=0; i<l.length; i++) {
            // note little-endian encoding - endianness is irrelevant as long as it matches longsToStr()
            l[i] = s.charCodeAt(i*4)        + (s.charCodeAt(i*4+1)<<8) +
                (s.charCodeAt(i*4+2)<<16) + (s.charCodeAt(i*4+3)<<24);
        } // note running off the end of the string generates nulls since bitwise operators treat NaN as 0
        return l;
    }


    /**
     * Converts array of longs to string.
     * @private
     */
    static longsToStr(l) {
        let str = '';
        for (let i=0; i<l.length; i++) {
            str += String.fromCharCode(l[i] & 0xff, l[i]>>>8 & 0xff, l[i]>>>16 & 0xff, l[i]>>>24 & 0xff);
        }
        return str;
    }


    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */


    /**
     * Encodes multi-byte string to utf8 - monsur.hossa.in/2012/07/20/utf-8-in-javascript.html
     */
    static utf8Encode(str) {
        return unescape(encodeURIComponent(str));
    }

    /**
     * Decodes utf8 string to multi-byte
     */
    static utf8Decode(utf8Str) {
        try {
            return decodeURIComponent(escape(utf8Str));
        } catch (e) {
            return utf8Str; // invalid UTF-8? return as-is
        }
    }


    /**
     * Encodes base64 - developer.mozilla.org/en-US/docs/Web/API/window.btoa, nodejs.org/api/buffer.html
     */
    static base64Encode(str) {
        if (typeof btoa != 'undefined') return btoa(str); // browser
        if (typeof Buffer != 'undefined') return new Buffer(str, 'binary').toString('base64'); // Node.js
        throw new Error('No Base64 Encode');
    }

    /**
     * Decodes base64
     */
    static base64Decode(b64Str) {
        if (typeof atob == 'undefined' && typeof Buffer == 'undefined') throw new Error('No base64 decode');
        try {
            if (typeof atob != 'undefined') return atob(b64Str); // browser
            if (typeof Buffer != 'undefined') return new Buffer(b64Str, 'base64').toString('binary'); // Node.js
        } catch (e) {
            throw new Error('Invalid ciphertext');
        }
    }

}


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

export default Tea;
```