DI Management Cryptography with International Character Sets

If you are having problems doing cryptography with international character sets, you need to understand just one thing: **"Work with bytes not text strings"**.

This is a complete re-write of the page we first wrote in 2004. This new version brings it up to date, adds lots more information, and fixes up some poor terminology and, er..., mistakes in the earlier version.

Introduction

The most common problem arises from using the wrong character encoding, and this is usually due to working with string types instead of byte arrays. We look at the "correct" way to encrypt a string, the similar problems with message digest hashes, and how to solve some of the problems and the two rules.

To work with different character encoding schemes you should understand the Unicode standard and its subsets ASCII and Latin-1, and be aware of the alternative MBCS and DCBS encodings. We look at the different encoding schemes in Unicode like UTF-8, give examples and take note of the Byte Encoding Mark. We examine the different results you will get when encrypting different character encodings and give some suggestions of what to do if the decrypted text won't display.

Contents

Cryptographic functions operate on bytes, not characters

All operations with modern encryption algorithms like Triple DES, AES, Blowfish and RSA, and message digest hash algorithms like SHA-1 and MD5 operate on a *sequence of bytes*, not on a string of characters. So the simple rule is always to convert your "String" type into an array of "Byte" types before passing it to the cryptographic function. And do not try to stuff ciphertext bytes back into a String type.

It is helpful to classify data into two different types when doing cryptographic operations: **'text'** and **'binary'**.

- **'Text'** consists of readable, printable characters we expect to see on our computer screen or in a printout. Computers store 'text' internally in several different formats and your programming language may use another format as well.
- **'Binary'** data is a sequence of bits (a *bit string*) that are stored in a computer as a sequence of 8-bit bytes (a *byte* is sometimes called an *octet* and a sequence of bytes is sometimes called an *octet string* or *byte array*).

A *character* is, strictly speaking, an abstract concept meaning the smallest component of a written language that has semantic value. We use the term here more loosely for something like "LATIN CAPITAL LETTER A" [**]. This character looks like

A, perhaps, or $\mathbb{A}$ or maybe A.

You know what it means when you see it. So does your computer's operating system when it comes across the byte sequence that represents that character. Your programming language may deal with characters internally in a different way, but it will still display and print them correctly.

[**] Actually, latin capital letter a is the description of a *code point* in Unicode.

A *string* is a sequence of characters, like "abc" or "Hello world!" Your computer will store this internally as a sequence of bytes in an encoding format that it knows. When you ask the computer to display this string on the screen or print it, it handles all the

necessary housekeeping tasks to make sure what you see is what you get.

The catch is that there are several different ways to encode a given character into a byte sequence. To pass the information about the string "ABC" to a cryptographic function like `Encrypt()` you need to convert this to a sequence of bytes. The correct term for this is to *encode* the characters. Then you need to *decode* the decrypted bytes back to characters.

In the old days, all programming was done with strings of simple ANSI characters each of which fitted into one byte. A "string" of characters would be stored as an array of single bytes in the same order and - to the computer - there was no difference between them. So English-speaking and western European programmers got used to using the term "character" and "byte" as meaning the same thing. Then they wrote programs that treated the string type as identical to an array of byte types and it seemed to work. Well, it worked until they tried to use it with international character sets.

We'll look below at the different ways to encode a character to a sequence of bytes, but first we will look at the steps to encrypt a string properly.

Encryption and Decryption

The processes of encrypting a string of textual characters and decrypting it again are both done in two distinct steps, as follows. We use the term **byte array** to mean a bit string encoded as a sequence of bytes. The function names are generic so don't go looking for them in your .NET documentation.

Algorithm: Encryption

INPUT: String of characters, `input_string`, and encryption `key`.
OUTPUT: Byte array of ciphertext, `ciphertext_bytes`.

1. Encode the string of textual characters to a byte array, perhaps with padding.
   `plaintext_bytes <-- EncodeBytesFromString(input_string)`
   This is the *plaintext* input to the encryption function.
2. Use the encryption function to transform the plaintext byte array to another byte array of *ciphertext*.
   `ciphertext_bytes <-- Encrypt(key, plaintext_bytes)`

Return `ciphertext_bytes`.

The output is forwarded to the recipient, who decrypts it. Perhaps the encrypted byte array is sent directly, or maybe saved as a binary file and that is forwarded, or maybe encoded again to a base64 string, which is much easier to handle.

Algorithm: Decryption

INPUT: Byte array of ciphertext, `ciphertext_bytes` and encryption `key`.
OUTPUT: String of characters, `output_string`.

1. Use the decryption function to transform the ciphertext byte array to a plaintext byte array.
   `plaintext_bytes <-- Encrypt(key, ciphertext_bytes)`
2. Decode the resulting byte array to a readable string of textual characters.
   `output_string <-- DecodeStringFromBytes(plaintext_bytes)`

Return `output_string`.

Some cryptography packages provide separate functions to do these two steps. Others do both steps in one go.

Message digest hashes and digital signatures

The same two-step approach applies when computing the *message digest* of a string using a *hash* like SHA-1 or MD5. You (1) encode the string of characters into an array of bytes; then (2) pass that array of bytes to the hash function. There is no reverse procedure (because it's a one-way hash).

Message digests are used to create a *digital signature* with *public key* algorithms like *RSA*. In effect the message digest is "encrypted" with the signer's private key. To verify the signature, the receiver uses the signer's public key to "decrypt" the signature value and then extracts the message digest value (H). Then she takes the original text and computes its message digest value (H'). If `H = H'` then the signature is valid; if not, then the signature is invalid.

There are variants of this like DSS, but in essence they all involve computing the message digest of the original text. Get that wrong and the signature will fail.

Problems and solutions

1. People do not bother to encode their strings to a byte array before encrypting or decode back after decryption because they are used to systems where the byte array used for encryption and the internal representation of the corresponding string of characters are identical. This is the case when using ASCII or ANSI characters on a system using a *Single Byte Character Set* (SBCS). Problems happen when they are not.

   Solution: Use bytes not strings. Take control over how your strings are encoded into bytes.

2. When the internal encoding of characters on the sender's system is different from that on the recipient's, the recipient's system will not be able to interpret the characters.

Solution: Sender encodes characters in the required format before encrypting; or recipient decodes decrypted bytes into correct format after decrypting.

3. The cryptographic function accepts a "string" as input but makes the assumption that each character is encoded as a single byte when it is not.

   Solution: Use a package that explicitly works with byte arrays (or accepts hex- or base64-encoded strings).

4. Someone tries to use the `DecodeStringFromBytes` function on the `ciphertext_bytes` array. Remember that ciphertext is a random sequence of bytes with no structure and so decoding back to a string may not work. At best it will give you unprintable garbage (unprintable in the sense that it displays as lots of funny characters and makes your computer beep). It may also change the values of the underlying bytes, so if you try and decrypt it later, it fails. Therefore be careful because

   ```
   EncodeBytesFromString(DecodeStringFromBytes(ciphertext_bytes))
   ```

   does not necessarily equal `ciphertext_bytes`.

   Solution: Don't try and store ciphertext bytes in an string type.

It does not help that the terms 'plaintext' and 'ciphertext' imply a string of characters. This is a throwback to days of character-based ciphers like the Caesar or Enigma ciphers where the ciphertext really was a string of characters. In modern ciphers we use byte arrays.

The Two Rules

So we have two rules:

Rule 1: **"Work with bytes not text strings"**

Rule 2: **"Do not put ciphertext bytes directly into a string type"**

How to store ciphertext bytes

Ciphertext byte arrays are quite inconvenient to handle. They can consist of any of the possible 256 bytes, most of which represent "funny" characters or control codes that make your PC beep. You can't print them, you can't paste them directly into an email, and as we've seen above, you can't put them *directly* back into a string type.

The *only* safe way to store a byte array of ciphertext directly is to save directly in a binary file. Make sure it is saved in "binary" mode. The file size after saving should be exactly the same as the length of the byte array. (Sometimes if files are saved in "text" mode, any LF characters will automatically be converted to CR-LF pairs, or end-of-file or Unicode BOM markers get added.) Remember when you come to decrypt, you must have exactly the same sequence of bytes as was output by the Encrypt function.

It is much easier is to encode the bytes into a "safe" string format (yes, here's that word encode again). Encode the byte array to a string of hexadecimal or base64 characters and use that string instead. This string you can print or paste into an email or just send direct to your recipient as normal text.

A hexadecimal (hex) string looks like `"FEDCBA9876543210"`. We show our examples in hex. A hex string uses two characters in the range [0-9A-Fa-f] to represent a byte value. There must always be an even number of characters and it doesn't matter whether the letters are upper- or lower-case. Hexadecimal strings are sometimes known as base16 strings. Hex strings are ideal for debugging because you can see the exact structure of the data. It does double the size, though.

A base64 string (called radix64 in PGP) looks like `"/ty6mHZUMhA="` or `"pnu8/1b+WWkzgoe8sbZhvk6QFz=="`. Valid base64 characters are [A-Za-z0-9/+] plus perhaps one or two padding "=" characters. The length should be divisible by 4 (but it isn't critical if it's not). Base64 strings are shorter than hex, but are impossible to use for debugging purposes. They are used, for example, in signed XML documents.

You can also change the formatting of these base64 or hex strings by adding newlines and perhaps those boundaries with BEGIN FOO and END FOO. This does not change the nature of the encoded data. This is another reason why these "safe" encoded strings are so useful.

So `"FEDCBA9876543210"` is the same as `"FE DC BA 98 76 54 32 10"` is the same as `"FE:DC:BA:98:76:54:32:10"` is the same as

```
FEDCBA98
76543210
```

and `"pnu8/1b+WWkzgoe8sbZhvk6QFz=="` is the same as

```
pnu8/1b+WWkzgo
e8sbZhvk6QFz==
```

is the same as

```
-----BEGIN FOO-----
pnu8/1b
+WWkzgo
e8sbZhv
k6QFz==
-----END FOO-----
```

once you have stripped off the "-----BEGIN FOO----" and "-----END FOO-----" encapsulation.

The problem with the word "encode"

The word "encode" is used here to mean transform, not encrypt. Yes, it is correct that "encrypt" is a type of encoding, but the "encoding" we're talking about doesn't add any security. So don't go thinking that encoding plaintext to base64-encoded form has made your information secret. It's just less readable.

Unicode, ANSI, ASCII and Latin-1

The term *Unicode* has several meanings:

1. The *Unicode Consortium*, the standards development organisation.
2. The standard for digital representation of the characters used in writing all of the world's languages.
3. A term used (not quite correctly) in older Microsoft documentation to mean a character-encoding scheme that uses 2 bytes for every character, as opposed to "MBCS" and "ANSI".

Think of the Unicode standard as a large codepage that maps every known character to a unique *codepoint* (see Unicode Code charts). It covers other issues, too, like rules to sort characters and how to join characters in cursive scripts. Most importantly, it defines seven character encoding schemes. Yes, seven. We discuss them below.

A Unicode codepoint is written in the form "U+XXXX" or "U+XXXXX" where [X]XXXX is the value of the codepoint in hexadecimal. Valid Unicode code point values are U+0000 to U+10FFFF. There is room for over one million characters (actually 1 114 112) of which under 100,000 have been allocated so far. Some examples of code points are:

- U+0041 - latin upper case letter a
- U+00E9 - latin small letter e with acute (é)
- U+3061 - hiragana letter ti ち
- U+4E2D - the CJK ideograph zhong 中
- U+1D160 - musical symbol eighth note 𝅘𝅥𝅯

The ASCII character set

The *ASCII* character set is the set of 128 characters from U+0000 to U+007F. See C0 Controls and Basic Latin Range: 0000-007F. It is a subset of Unicode. Each character can be encoded with just 7 bits. It includes the usual letters you find on a standard US keyboard, A-Z and a-z, digits 0-9 and the punctuation characters like !@#$%*, plus some *control codes* like "Carriage Return" and "Tab" which hark back to the days of manual typewriters and teletypes. It is a *single-byte character set* (SBCS) and the characters encode to the same "value" as the code point. So U+0041 latin upper case letter a encodes to the single byte with value 0x41 and U+007D right curly bracket encodes to the single byte with value 0x7D. Easy, eh?

Latin-1 and Windows-1252

Latin-1 is an alternative name for the western European *ISO-8859-1* character set. It is a SBCS whose code points match exactly those of Unicode, so it is also a subset of Unicode. The first 128 characters are the same as ASCII. The codepoint U+00E9 latin small letter e with acute (é) is in Latin-1 and is encoded by the single byte of value 0xE9. *Windows-1252* is the default code page on most Windows computers. It is identical to ISO-8859-1 except for the block of 32 characters between U+0080 and U+009F which is reserved for control codes in ISO-8859-1 but is used to add some extra printable characters in Windows-1252 (like the double quotation marks “ and ” and the euro symbol €).

The ANSI character set

The *ANSI character set* is an old and incorrect Microsoft term for all Windows code pages. It is a single-byte character set where the first 128 characters are (usually) the same as ASCII and the next 128 from 0x80 to 0xFF are used for many other characters including accented characters, various symbols, and those characters you can use to draw boxes on the console. The actual character in the upper range depends on the active code page in the PC. Code pages are a pain to use but the underlying byte values do not change if you change the code page, it just displays differently. In MSVC, if you are in "ANSI" mode it means all characters you are using can be encoded in a single byte.

As a general rule, if the byte does not have its high bit set (i.e. is in the range 0x00 to 0x7F) then it represents an ASCII character (there may be some obscure code pages where this isn't so). If not, then it represents whatever character the current code page says it does.

MBCS and DBCS

The term "MBCS" means *multi-byte character set* and can be applied to any character set that needs more than one byte to encode its characters. By that definition, Unicode is a multi-byte character set. However, in the Microsoft world, "MBCS" is used to mean a particular character set used to represent "international" characters using a system of lead bytes.

In general, the first 128 characters in the range 0x00 to 0x7F are identical to the ASCII character set. Certain other bytes are designated as *lead bytes* which mean the bytes following define the character. The lead byte designates the code page to use. For example, the lead byte 0x82 indicates that the character is from the Hiragana set and can be found in the See Microsoft Windows Codepage:932(Japanese Shift-JIS) with Lead Byte=0x82.

In principle, a MBCS can be encoded with several bytes, making it capable of representing an unlimited number of characters. But in Microsoft Visual C++, MBCS is limited to two bytes, making it a *double-byte character set* (DBCS). The terms "MBCS" and "DBCS" mean the same thing in Microsoft documentation.

So, in Microsoft-Visual-C-world, we have three environments:

ANSI
: A superset of ASCII with a single-byte character set, the exact characters depending on the code page in operation on the

user's machine.

MBCS (or DBCS)
: A character set that uses 1 or 2 bytes to represent a character, allowing more than 256 characters to be represented. The lead byte indicates the codepage to use.

UNICODE
: Uses the (Microsoft variant of the) Unicode standard. Internally, Windows uses a UTF-16LE character encoding scheme.

Remember that a compiler like MSVC++ might offer you different character sets from the one the underlying operating system uses itself.

Character encoding schemes and the Unicode Transformation Format (UTF)

Unicode has three *character encoding forms* known as *Unicode Transformation Formats* (UTF) which map code points to a series of fixed-length bit patterns called *code units*. There are then seven *character encoding schemes* (also ambiguously known as UTFs) which map these code units to byte sequential order. Got that?

```
code point (position in table) --> code unit (bit pattern) --> byte sequence
```

In Unicode we have seven *character encoding schemes*

- UTF-32
- UTF-32BE
- UTF-32LE
- UTF-16
- UTF-16BE
- UTF-16LE
- UTF-8

Unicode used to have *UCS-2* which encoded every character in exactly two bytes. At that time Unicode expected to have less than 0xFFFF (65535) code points, so all could be encoded in two bytes. Older versions of Windows NT before XP and programming languages like VB6 use UCS-2. UTF-16 is an extension that uses *surrogate pairs* to extend the set of possible code points from 65535 to just over one million. In practice, all characters used in commerce have code points below U+FFFF and can be encoded in two bytes. You only need surrogate pairs for really obscure characters like musical symbols or Egyptian hieroglyphs. But the catch for programmers is that, with UTF-16, you can no longer guarantee when parsing a byte sequence that two bytes always represent exactly one character.

UTF-32

UTF-32 uses a 32-bit code unit which is big enough to store all the maximum possible 21-bit code point values (The Unicode organisation has promised never to exceed this limit of just over 1 million - well, at least for the moment). There are three schemes associated with this form

UTF-32BE
: a "big-endian" version that serializes each code unit most-significant byte first.

UTF-32LE
: a "little-endian" version that serializes each code unit least-significant byte first.

UTF-32
: strictly a self-describing version that uses an extra sentinal value at the beginning of the stream, called the *byte order mark* (BOM), to specify whether the code units are in big-endian or little-endian order. In practice, the term "UTF-32" is generally used to mean the form not the scheme with the byte order being assumed in the context. We'd always say something like "UTF-32 with BOM" to make sure.

UTF-32 is convenient from a programming point of view, as every character is encoded to four bytes, but wasteful of space. Linux systems use UTF-32 internally.

UTF-16

UTF-16 represents each 21-bit code point value as a sequence of one or two 16-bit code units. There are three similar schemes associated with this form: UTF-16BE, UTF-16LE and UTF-16 with a BOM.

UTF-16 can represent the vast majority of characters in the world with a single 16-bit code unit. It is more efficient for storing CJK characters than UTF-8.

UTF-8

UTF-8 represents each 21-bit code point value as a sequence of one to four 8-bit code units. It has the big advantage that it is backwards compatible with ASCII and doesn't break programs written in C. It is more efficient than UTF-16 for storing western characters. It is used widely in HTML and XML documents. Older programming languages like VB6 do not "understand" UTF-8.

UTF-8 does not have or need little-endian or big-endian variants.

> The terms **big-endian** and **little-endian** come from Jonathan Swift's satire *Gulliver's Travels* written in 1726 in which the Big Endians were a political faction that broke their eggs at the large end and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

Examples of different character encodings

Encoding scheme	U+0041 latin upper case letter a	U+00E9 latin small letter e with acute	U+3061 - hiragana letter ti	U+1D160 - musical symbol eighth note

ASCII	41	-	-	-
Latin-1	41	E9	-	-
UTF-16BE	00 41	00 E9	30 61	D8 74 DD 60
UTF-16LE	41 00	E9 00	61 30	74 D8 60 DD
UTF-32BE	00 00 00 41	00 00 00 E9	00 00 30 61	00 01 D1 60
UTF-32LE	41 00 00 00	E9 00 00 00	61 30 00 00	60 D1 01 00
UTF-8	41	C3 A9	E3 81 A1	F0 9D 85 A0

Byte Order Mark (BOM)

The *Byte Order Mark* (BOM) is the Unicode character U+FEFF (ZERO WIDTH NO-BREAK SPACE) used as an optional sentinel to indicate the byte order of a text. If a program on a big-endian machine reads little-endian-encoded text, it will read this the wrong-way-round as U+FFEE, which is illegal in Unicode. Therefore the program should understand it has data in little-endian byte sequence order and act accordingly. The character U+FEFF read correctly will have no effect on the text displayed, being a zero-width no-break space - a term also used by many wives to describe their husbands.

The byte order marks encode as

Unicode scheme	Byte order mark encoding
UTF-16BE	FE FF
UTF-16LE	FF FE
UTF-32BE	00 00 FE FF
UTF-32LE	FE FF 00 00
UTF-8	EF BB BF

The BOM is not really needed for UTF-8 because it is possible to recognize a byte sequence that represents valid UTF-8 text by a simple algorithm. A BOM is added, though, to UTF-8 files by Notepad and by MS Visual Studio. Which is why you may find the odd characters ï»¿ when you view the file with another application or as ´╗┐ on the command-line console.

Of course, the presence or not of a BOM makes a huge difference to the result of cryptographic operation. The sender and recipient would need to agree whether or not to include it. This causes problems in particular when trying to sign an XML file using the ghastly *XML-DSIG* standard (the correct procedure is to remove it before computing the signature).

Effect of different character encoding when encrypting

Let's say we want to encrypt the 6-character string "Hello!"; that is, the sequence of 6 characters with codepoints

`U+0048 U+0065 U+006C U+006C U+006F U+0021`

As a simple example, we will use DES encryption in ECB mode with PKCS#5/7 padding with the key `0xFEDCBA9876543210`. (*Please note that single DES is no longer considered secure and ECB mode is definitely not secure*. We use it here as an illustration to keep the encrypted examples shorter. In practice, use at least Triple DES or AES-128 in CBC or CTR mode.)

Input string = "Hello!"

Encoding scheme	Encoded bytes	Resulting ciphertext bytes
ASCII/Latin-1	48656C6C6F21	7E5856F0CF6E3AB0
UTF-16BE	00480065006C006C006F0021	BFB8CF02A0E0D01113B693128BFE6CC6
UTF-16BE+BOM	FEFF00480065006C006C006F0021	D062C6425E03615A2BCE3AD56A109488
UTF-16LE	480065006C006C006F002100	8CE18992E3558713C6A97E4009F610E6
UTF-16LE+BOM	FFFE480065006C006C006F002100	86416B4D51342B0DB7A91D9F12E58E45
UTF-32BE	00000048000000650000006C0000006C0000006F00000021	60C8981589CF104AD92F5130B15448CFDE0F44823B06182CA2
UTF-32LE	48000000650000006C0000006C0000006F00000021000000	40F8233F22592213D524148E50927B4FF8C8B8E6F6386161A2A
UTF-8	48656C6C6F21	7E5856F0CF6E3AB0

Look how different the ciphertext is for each different type of encoding. Remember this is all for the *same* text string. (It's also a graphic example of how wasteful UTF-32 encoding is.) We used our CryptoSys API test bed demo program to generate these values (see image and note the option settings we used)

Hello in Japanese

In Hiragana Japanese characters, the word 'Hello' is KO-N-NI-TI-HA (Kon'nichiwa) こんにちは (graphic version).

The five characters have code points U+3053 (hiragana letter ko), U+3093 (hiragana letter n), U+306B (hiragana letter ni), U+3061 (hiragana letter ti), and U+306F (hiragana letter ha). Encrypting a selection of different encodings using the same DES key `0xFEDCBA9876543210` as above, we get

Input string = KO-N-NI-TI-HA

Encoding scheme	Encoded bytes	Resulting ciphertext bytes
UTF-16BE	30533093306B3061306F	EB0A6B3601461EB46CCCFD4C31F76A79

UTF-16LE	6F30533093306B306130	18BF045D0E2AB3D3CDB69BDF3373A81A
UTF-8	E38193E38293E381ABE381A1E381AF	ED7514181029A993B383B0194F172E1F
MBCS Shift-JIS	82B182F182C982BF82CD	A76ECB9C63DFF2B770567EAD9B72A9E8

Note how different the encoded bytes are *before* encrypting.

What to do if the decrypted text won't display

It's not as bad as it seems. If you are the recipient and are using the same key, the value of `plaintext_bytes` will be the same as that produced by the sender. It just doesn't display on your system when you convert the bytes to text. Perhaps you are set up for Shift-JIS Japanese characters and the sender's system uses UTF-16, or yours is a big-endian machine and theirs is little-endian.

But there will be some structure in the ciphertext bytes and you can use a hex editor to examine it (we use the FRHED free hex editor).

- Is every second byte zero? If so, it's probably UTF-16. Look at the encoded bytes for "Hello!" in UTF-16BE:

  ```
  00 48 00 65 00 6C 00 6C 00 6F 00 21
  ```

- Is it mostly zeros with only every fourth byte not? It's probably UTF-32.
- Is there a BOM? Should there be one and it's missing?
- Is there some other byte that repeats mostly every second one? If you are expecting data in Japanese note that Characters in Hiragana have Unicode code points of the form U+30xx, and you can see in the encodings above that every second byte is 0x30, so that indicates UTF-16 encoding. Similarly the lead byte in Shift-JIS is 0x82 and you can see that every other byte above is 0x82. UTF-8 is a bit harder, but Hiragana characters are encoded into 3-byte sequences beginning with 0xE3. You get the idea?
- Is it completely random? If you seem to have completely random bytes that include lots of the "funny" 8-bit ANSI characters and the occasional zero byte, then it's most likely you've used the wrong key to decrypt. In particular the presence of zero bytes is a good indication it's not encoded text.

With a bit of experience of how these encoded byte arrays work, you should be able to figure out what encoding has been used. Then you can use the appropriate encoding transformation to get the correct character string for your system.

For more information on converting strings to bytes and vice versa in programming different languages see Storing and representing ciphertext.

Summary

Work with bytes not text strings

The input to an encryption process must be 'binary' data. We need to convert the text we want to encrypt into 'binary' format first and then encrypt it. The results of encryption are *always* binary. Do not attempt to treat raw ciphertext as 'text' or put it directly into a `String` type. Store ciphertext either as a raw binary file or convert it to base64 or hexadecimal format. You *can* safely put data in base64 or hexadecimal format in a `String`.

When you decrypt, always start with binary data, decrypt to binary data, and then and only then, convert back to text, if that is what you are expecting. You can devise your own checks to make sure the decrypted ciphertext is what you expect before you do the final conversion.

Do not put ciphertext bytes directly into a string type

Don't try and store encrypted bytes in a string type.

When you pass ciphertext data between systems, we strongly recommend that you do *not* pass the data as raw 'binary' data. Binary data can easily be corrupted in transit, and a different systems may interpret some of the bytes in the binary data as control characters with unintended results. This is before we deal with the endian-ness of the two systems.

You should *always* encode the ciphertext data using, say, base64 or hexadecimal encoding. The resulting text can be easily transferred without loss of integrity and can then be decoded back into binary on the destination system before decryption.

We usually use hexadecimal encoding for short messages because it is easy to see the results and count the bytes. It is virtually impossible to decode base64 data in your head, even though it's shorter, and this is a major disadvantage when debugging. Just make sure that the other party agrees to use the same encoding as you do.

References

- Richard Gillam, *Unicode Demystified*, Addison-Wesley, 2002. You can view the first chapter on Amazon ISBN 0201700522.
- James Brown, Introduction to Unicode. A very clear explanation.
- John Skeet, Unicode and .NET. From his excellent C# in Depth book.
- Joel Spolsky, The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets.
- Jucca Korpela, A tutorial on character code issues. Very detailed.
- Angel Ortega, Unicode, UTF-8 and all that. Especially useful for C programmers wanting to convert between MBCS and Unicode.
- Ken Lunde, *CJKV Information Processing*, O'Reilly, 1999. The first chapter can be downloaded from <ftp-link>.
- Ken Lunde, Perl & Multiple-byte Characters, CJKV Type Development, Adobe Systems Incorporated, 1997.
- RFC 3629, UTF-8, a transformation format of ISO 10646. The nuts, bolts and bits.

Related Topics

See also our pages on:

- Cross-Platform Encryption
- Storing and representing ciphertext
- Using Byte Arrays in Visual Basic VB6
- Binary and byte operations in Visual Basic VB6

Contact

For more information or to comment on this page, please send us a message.

This page last updated 15 December 2012
