How to Calculate the Size of Encrypted Data?

If you need to store ciphertext in a database table (or array variable), you may need to know how long a ciphertext value can be so you can allocate enough memory to hold it. This article explains how to calculate the size of ciphertext produced by different cryptographic algorithms. In addition, it highlights the issues associated with the ciphertext storage and briefly describes several aspects of cryptography, which are essential for understanding the topic. Please be aware that the article intentionally oversimplifies certain concepts to make them easier to understand for novice programmers. For additional information about the relevant subjects see References.

Hashing

Hashing is commonly used for password-based authentication. Calculating the size of hashed data is easy because it does not change for a given hashing algorithm (or *hash function*). Each hashing algorithm generates the same number of bytes irrespective of the plaintext size. It does not matter whether you hash a one-character string or a thousand-character string; as long as you use the same hashing algorithm, the size of the resulting hash values will always be the same. Table 1 defines the hash sizes produced by different hashing algorithms.

Table 1. Hash sizes

Hashing algorithm	Hash size
MD5*	16 bytes (128 bits)
SHA-1	20 bytes (160 bits)
SHA-256	32 bytes (256 bits)
SHA-384	48 bytes (384 bits)
SHA-512	64 bytes (512 bits)
* The MD5 hashing algorithm is not recommended.	

If you use hashing with *salt* (as you should), you will need to store the salt values along with the ciphertext. You can either store the salt values in a separate column or appended them to the generated hashes; since the length of the hash values is fixed, your program can easily extract salt from the stored ciphertext. If you decide to store salt appended to the ciphertext, do not forget to increase the size of the column holding it.

Encryption

The size of ciphertext produced by two-way encryption depends primarily on two factors: (a) length of the plaintext value and (b) type of encryption algorithm. With respect to their effects on the size of encrypted data, encryption algorithms can be categorized as *stream ciphers* and *block ciphers*

Stream ciphers

Encryption algorithms using stream ciphers (such as RC4) encrypt plaintext sequentially, one bit at a time. The size of the resulting ciphertext should be exactly the same as the original plaintext value, which makes this case even more trivial than hashing. (Note: In general, you should avoid using stream ciphers.)

Block ciphers

Better encryption algorithms use block ciphers. These algorithms include symmetric-key algorithms, such as Rijndael (also known as AES), Triple-DES, and RC6, as well as public-key algorithms, such as RSA. The algorithms using block ciphers encrypt several bits of plaintext data in one step. The bits encrypted in one step make up what is called an *encryption block*. If an encryption algorithm uses block ciphers, the size of the ciphertext will always be a multiple of the encryption block size.

Encryption block sizes vary between different encryption algorithms and may also vary within the same encryption algorithm. For example, the Rijndael algorithm can use 128, 192, or 256-bit blocks. It is worth noting that not all block sizes supported by the algorithm specification, may be provided in every implementation. For instance, the Microsoft Cryptographic Service Provider (CSP) version of the Rijndael algorithm only supports 128-bit blocks, while the .NET implementation (RijndaelManaged) supports all three options.

Padding

If the size of the original plaintext encrypted using a block-cipher algorithm is not an exact multiple of the block size, *padding* will be used to make up the difference. Padding means appending additional (mostly meaningless) bytes at the end of the plaintext data before performing encryption. Because after decrypting data the algorithm must know how many bytes of padding must be removed from the result, the padding bytes must also contain certain information (metadata) about these bytes. For example, when the PKCS #5 padding is used, each padding byte will contain the value indicating the total number of padding bytes (see Figure 1).

Figure 1. PKCS #5 padding

H	e	l	l	o	,		w	o	r	l	d	!	0x3	0x3	0x3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

You do not need to know much about padding except that it may increase the size of the ciphertext. If the size of the original data (encrypted using a block-cipher algorithm with padding) is an exact multiple of the encryption block size, padding will increase the size of the ciphertext by one block. In case of public-key encryption, padding will decrease the maximum length of the plaintext data that can be encrypted with a key of certain length.

Examples

Now that you are familiar with the terminology, let us do some math. To calculate the size of the ciphertext produced by the block-cipher encryption, you will need the following information:

a. Length of the plaintext value
b. Encryption block size
c. Padding information (if padding is used)

In the most generic case, the size of the ciphertext can be calculated as:

CipherText = PlainText + Block - (PlainText MOD Block)

where *CipherText*, *PlainText*, and *Block* indicate the sizes of the ciphertext, plaintext, and encryption block respectively. Basically, the resulting ciphertext size is computed as the size of the plaintext extended to the next

block. If padding is used and the size of the plaintext is an exact multiple of the block size, one extra block containing padding information will be added.

Let's say that you want to encrypt a nine-digit Social Security Number (SSN) using the Rijndael encryption algorithm with the 128-bit (16-byte) block size and PKCS #7 padding. (For the purpose of the illustration, assume that dashes are removed from the SSN value before the encryption, so that "123-45-6789" becomes "123456789", and the value is treated as a string, not as a number.) If the digits in the SSN are defined as ASCII characters, the size of the ciphertext can be calculated as:

$$\text{CipherText} = 9 + 16 - (9 \text{ MOD } 16) = 9 + 16 - 9 = 16 \text{ (bytes)}$$

Notice that if the size of the plaintext value is the exact multiple of the block size, an extra block containing padding information will be appended to the ciphertext. For example, if you are to encrypt a 16-digit credit card number (defined as a 16-character ASCII string), the size of the ciphertext will be:

$$\text{CipherText} = 16 + 16 - (16 \text{ MOD } 16) = 16 + 16 - 0 = 32 \text{ (bytes)}$$

Unicode

To determine the size of an encrypted string, you must consider the character format. Because most encryption routines operate on byte arrays, you have to know how to convert characters to bytes. In the simplest case of ASCII strings, the number of characters and the number of bytes are the same. If the string contains *Unicode* characters, you have to adjust the size accordingly. In a typical (but not necessarily every) case, one Unicode character uses two bytes (see Figure 2 and Figure 3).

Figure 2. "Hello!" in ASCII

H	e	l	l	o	!
0	1	2	3	4	5

Figure 3. "Hello!" in Unicode

H	\0	e	\0	l	\0	l	\0	o	\0	!	\0
0	1	2	3	4	5	6	7	8	9	10	11

If a nine-digit Social Security Number (SSN) were a nine-character Unicode string, the size of the resulting ciphertext would have been calculated as:

$$\text{CipherText} = (9 \times 2) + 16 - ((9 \times 2) \text{ MOD } 16) = 18 + 16 - 2 = 32 \text{ (bytes)}$$

String terminator

Depending on the programming language you use, you may need to decide whether to encrypt the end-of-string character (NULL or '\0') along with the plaintext string value (see Figure 4).

Figure 4. End-of-string character (ASCII)

H	e	l	l	o	!	\0
0	1	2	3	4	5	6

Most popular .NET languages (such as C# or Visual Basic.NET) perform the conversion between strings and byte arrays automatically, so you do not need to worry about string terminators; however, if you use C or C++, a reasonable option would be to include the string terminator when converting a string to a byte array. This will increase the size of the plaintext by one (for ASCII strings) or two (for Unicode strings) bytes. If you do not encrypt the string terminator, after performing decryption you will need to add it explicitly.

Trailing nulls

If you are planning to store encrypted data in a database table in binary format, you must be aware that Microsoft® SQL Server™ can trim trailing nulls of the stored data. (This feature is controlled by by the value of the ANSI_PADDING database configuration option.) Because ciphertext can contain trailing nulls, it may be a good idea to append a non-zero byte at the end of all encrypted values before storing them in the database (and discard it before performing decryption). Do not forget to account for this byte when calculating the size of the column.

Base64 encoding

Instead of storing encrypted values as binary data (byte arrays) and having to deal with trailing zeros, you can convert them to strings. Strings are also easier to handle if you need to store them in text files, such as application configuration files. Unfortunately, you cannot use regular strings to store ciphertext because it can contain unprintable characters. *Base64 encoding* solves this problem.

In a nutshell, base64 encoding converts a byte array into a printable ASCII string, which can be later transformed back to the original array. This is how base64 encoding works. The base64 encoding algorithm takes every three bytes of data and converts them into four bytes of printable ASCII characters. If the size of the incoming byte array is not an exact multiple of three, the algorithm appends equal signs (one for each missing byte) at the end of the base64-encoded string value. This convention guarantees that the size of base64-encoded string will always be a multiple of four. The length of a base64-encoded string can be calculated as:

$$Base64 = (Bytes + 2 - ((Bytes + 2) \text{ MOD } 3)) / 3 * 4$$

where *Base64* and *Bytes* indicate the number of bytes in the base64-encoded string and the original byte array respectively. You can use this formula to calculate the size of the column holding base64-encoded ciphertext.

For example, if a byte array contains 13 characters of the ASCII string "Hello, world!", the size of the corresponding base64-encoded string can be calculated as:

$$Base64 = (13 + 2 - ((13 + 2) \text{ MOD } 3)) / 3 * 4 = 20 \text{ (bytes)}$$

The resulting value will be "SGVsbG8sIHdvcmxkIQ==". The last two characters of the base64-encoded string contain two equal signs ("==") indicating the two missing bytes in the last three-byte block of the byte array. (Note: Please keep in mind that base64 encoding has nothing to do with data protection; it is not an encryption algorithm, but a simple encoding scheme, which can be applied to the already encrypted values.)

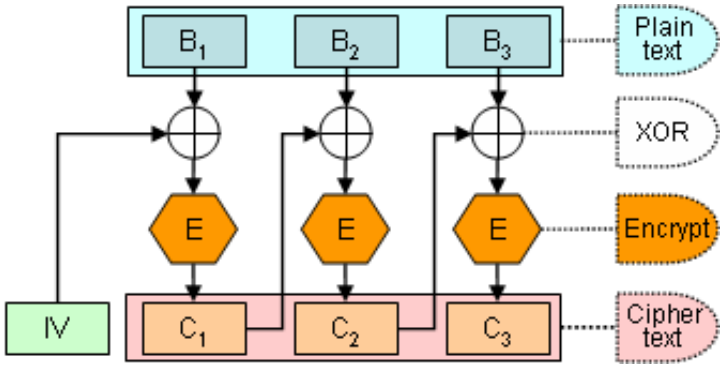
Initialization vector

While *initialization vector (IV)* does not affect the size of encrypted data, it may need to be considered when you think about encrypted data storage. Initialization vector is used by encryption algorithms (such as Rijndael,

Triple-DES, and others), which support *cipher-block chaining* (CBC) or a similar feedback mode, in which the result of encryption of each data block depends on the value of the previous block. When two identical plaintext values are encrypted with the same encryption key but different initialization vectors, the resulting ciphertext values will be different. (Note: The CBC mode is considered more secure than encryption modes, in which all data blocks are encrypted independently of each other, such as *electronic codebook*, or ECB.)

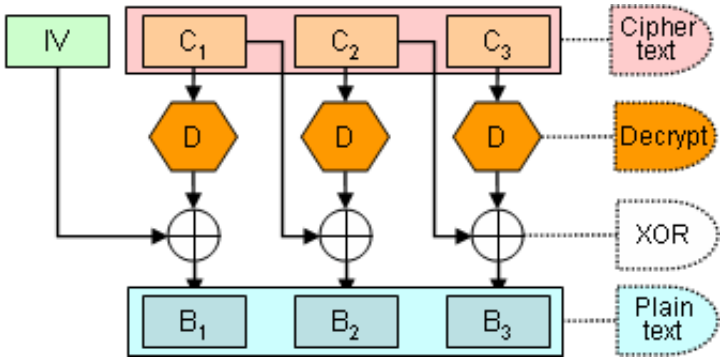
The problem with cipher-block chaining is that when the first block of plaintext data is to be encrypted, there is no previous block, so an initialization vector is used instead. This is how initialization vector works in the cipher-block chaining mode. Before being encrypted, the first block of the plaintext data is XORed with the initialization vector and the result of the XOR operation is encrypted producing the first encrypted block. This encrypted block is then XORed with the second plaintext block and the result is encrypted producing the second encrypted block. The rest of the encryption blocks are processed in a similar manner (see Figure 6).

Figure 6. Encryption in the CBC mode



The decryption routine in the CBC mode works almost the same as encryption (see Figure 7).

Figure 7. Decryption in the CBC mode



Because the initialization vector is required during decryption, it must be stored along with the encrypted data. (Note: Unlike the encryption key, the initialization vector contains no secret, so there is no need to protect it, although protecting it would not hurt.) Similar to salt used in hashing, the initialization vector can be stored in a separate table column or appended to the ciphertext, in which case you must consider it in the size calculations; because the size of the initialization vector is likely to be constant (normally, it must be the same as the size of the encryption block), it can be easily extracted from the ciphertext value.

Salt

Having to maintain different initialization vectors (with the same encryption key) for encrypted values stored in the

database may become a hassle. A reasonable alternative could be using the same initialization vector but appending a randomly generated salt at the beginning of the plaintext value before encrypting data (this is similar to hashing with salt). If you follow this approach and encrypt data with salt, include the length of the salt value along with the metadata needed by the decryption routine (which will need to know how many bytes of salt to discard before returning the plaintext value) in the ciphertext size calculations.

Public-key encryption

Although public-key encryption algorithms (such as RSA) also use block ciphers, their analysis requires a different approach than symmetric-key algorithms. First of all, you must realize that unlike encryption with symmetric key, which can be performed on plaintext of an arbitrary size, the maximum number of plaintext bytes that can be encrypted using a public key depends on the size of the key (and the type of padding).

For example, when using a 1024-bit RSA key with PKCS #1 v.1.5 padding (one of the most common options), you will not be able to encrypt a string, which is longer than 117 bytes (that is 117 ASCII or 58 two-byte Unicode characters). Increasing the size of the RSA key to 2048 bits will allow you to encrypt 245 bytes of data, but longer RSA keys are expensive: they take more time to generate and operate. Using long public keys (longer than 1024 bits) can seriously degrade application performance. (Note: You can encrypt longer data values using shorter public keys by splitting these values into the smaller blocks and encrypting them individually, but this approach is also not efficient and therefore cannot be recommended.)

Explaining how to calculate the maximum size of plaintext data encrypted using a public key deserves a separate article. It would require the explanation of such concepts and terms as exponent, modulus, primes, factoring, data signing, and so on. If you think that life is too short to waste on learning such grizzly details, just use the most reliable trial-and-error approach: pick the longest data value your application may need to use and try to encrypt it with a public key of a reasonable size (1024-bit key is considered optimal). Check the length of the encrypted array and you will have all the information you need. Even better, do not use public keys for encrypting long data. Because public-key encryption is designed primarily for key exchange and data signing, generally, it is not suited for traditional data encryption.

Conclusion

When designing a database table for storing encrypted (or hashed) data, consider the encryption method the program will use along with the data format and transformations. Although you can use simple calculations to determine the size of memory for holding the ciphertext, always test your results using the longest plaintext value supported by the application just to be on the safe side.

References

Base64 Explained
Cryptographic Tools (public/symmetric keys, stream/block ciphers, hash functions, etc.)
Encryption and Security Tutorial
How do I estimate the size of encrypted/decrypted data?
Initialization Vector
The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
Unicode and .NET