

Re: Encrypted network communication

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2004-09/2775.html>

From: Bredal Jensen (*Bredal.Jensen_at_mimosa.com*)

Date: 09/12/04

Date: Sun, 12 Sep 2004 10:13:41 +0200

Read the MSDN document below. I think a public key algorithm such as (RSA) which is available for use in DOTNET should help you accomplish this.

Cryptography protects data from being viewed or modified and provides secure channels of communication over otherwise insecure channels. For example, data can be encrypted using a cryptographic algorithm, transmitted in an encrypted state, and later decrypted by the intended party. If a third party intercepts the encrypted data, it will be difficult to decipher the data.

In a typical situation where cryptography is used, two parties (Alice and Bob) communicate over an insecure channel. Alice and Bob want to ensure that their communication remains incomprehensible by anyone who might be listening. Furthermore, because Alice and Bob are in remote locations, Alice must be sure that the information she receives from Bob has not been modified by anyone during transmission. In addition, she must be sure that the information really does originate from Bob and not someone impersonating Bob.

Cryptography is used to achieve the following goals:

- a.. Confidentiality: To protect a user's identity or data from being read.
- b.. Data integrity: To protect data from being altered.
- c.. Authentication: To assure that data originates from a particular party.

In order to achieve these goals, Alice and Bob use a combination of algorithms and practices known as cryptographic primitives to create a cryptographic scheme. The following table lists the cryptographic primitives and their uses.

Cryptographic Primitive Use

Secret-key encryption (symmetric cryptography) Performs a transformation on data, keeping the data from being read by third parties. This type of encryption uses a single shared, secret key to encrypt and decrypt data.

Public-key encryption (asymmetric cryptography) Performs a

transformation on data, keeping the data from being read by third parties. This type of encryption uses a public/private key pair to encrypt and decrypt data.

Cryptographic signing Ensures that data originates from a specific party by creating a digital signature that is unique to that party. This process also uses hash functions.

Cryptographic hashes Maps data from any length to a fixed-length byte sequence. Hashes are statistically unique; a different two-byte sequence will not hash to the same value.

Secret-Key Encryption

Secret-key encryption algorithms use a single secret key to encrypt and decrypt data. You must secure the key from access by unauthorized agents because any party that has the key can use it to decrypt data. Secret-key encryption is also referred to as symmetric encryption because the same key is used for encryption and decryption. Secret-key encryption algorithms are extremely fast (compared to public-key algorithms) and are well suited for performing cryptographic transformations on large streams of data.

Typically, secret-key algorithms, called block ciphers, are used to encrypt one block of data at a time. Block ciphers (like RC2, DES, TripleDES, and Rijndael) cryptographically transform an input block of n bytes into an output block of encrypted bytes. If you want to encrypt or decrypt a sequence of bytes, you have to do it block by block. Because the size of n is small ($n = 8$ bytes for RC2, DES, and TripleDES; $n = 16$ [the default]; $n = 24$; or $n = 32$ bytes for Rijndael), values larger than n have to be encrypted one block at a time.

The block cipher classes provided in the base class library use a chaining mode called cipher block chaining (CBC), which uses a key and an initialization vector (IV) to perform cryptographic transformations on data. For a given secret key k , a simple block cipher that does not use an initialization vector will encrypt the same input block of plaintext into the same output block of ciphertext. If you have duplicate blocks within your plaintext stream, you will have duplicate blocks within your ciphertext stream. If an unauthorized user knows anything about the structure of a block of your plaintext, she can use that information to decipher the known ciphertext block and possibly recover your key. To combat this problem, information from the previous block is mixed into the process of encrypting the next block. Thus, the output of two identical plaintext blocks is different. Because this technique uses the previous block to encrypt the next block, an IV is used to encrypt the first block of data. Using this system, common message headers that might be known to an unauthorized user cannot be used to reverse engineer a key.

One way to compromise data encrypted with this type of cipher is to perform an exhaustive search of every possible key. Depending on the size of the key used to perform encryption, this type of search is extremely time consuming using even the fastest computers and is therefore unfeasible. Larger key sizes are more difficult to decipher. Though encryption does not make it theoretically impossible for an adversary to retrieve the encrypted data, it

does raise the cost of doing so prohibitively. If it takes three months to perform an exhaustive search to retrieve data that is only meaningful for a few days, then the exhaustive search method is impractical.

The disadvantage of secret-key encryption is that it presumes two parties have agreed on a key and IV and communicated their values. Also, the key must be kept secret from unauthorized users. Because of these problems, secret-key encryption is often used in conjunction with public-key encryption to privately communicate the values of the key and IV.

Assuming that Alice and Bob are two parties who want to communicate over an insecure channel, they might use secret-key encryption as follows. Both Alice and Bob agree to use one particular algorithm (Rijndael, for example) with a particular key and IV. Alice composes a message and creates a network stream on which to send the message. Next she encrypts the text using the key and IV, and sends it across the Internet. She does not send the key and IV to Bob. Bob receives the encrypted text and decrypts it using the previously agreed upon key and IV. If the transmission is intercepted, the interceptor cannot recover the original message because he doesn't know the key or IV. In this scenario, the key must remain secret, but the IV does not need to remain secret. In a real world scenario, either Alice or Bob generate a secret key and use public-key (asymmetric) encryption to transfer the secret (symmetric) key to the other party. For more information, see Public-Key Encryption.

The .NET Framework provides the following classes that implement secret key encryption algorithms:

- a.. DESCryptoServiceProvider
- b.. RC2CryptoServiceProvider
- c.. RijndaelManaged
- d.. TripleDESCryptoServiceProvider

Public-Key Encryption

Public-key encryption uses a private key that must be kept secret from unauthorized users and a public key that can be made public to anyone. Both the public key and the private key are mathematically linked; data encrypted with the public key can only be decrypted with the private key and data signed with the private key can only be verified with the public key. The public key can be made available to anyone; this key is used for encrypting data to be sent to the keeper of the private key. Both keys are unique to the communication session. Public-key cryptographic algorithms are also known as asymmetric algorithms because one key is required to encrypt data while another is required to decrypt data.

Public-key cryptographic algorithms use a fixed buffer size whereas secret-key cryptographic algorithms use a variable length buffer. Public-key algorithms cannot be used to chain data together into streams the way secret-key algorithms can because only small amounts of data can be encrypted. Therefore, asymmetric operations do not use the same streaming model as symmetric operations.

Two parties (Alice and Bob) might use public–key encryption as follows. First, Alice generates a public/private key pair. If Bob wants to send Alice an encrypted message, he asks her for her public key. Alice sends Bob her public key over an insecure network and Bob uses this key to encrypt a message. (If Bob received Alice's key over an insecure channel, such as a public network, Bob must verify with Alice that he has a correct copy of her public key.) Bob sends the encrypted message to Alice and she decrypts it using her private key.

During the transmission of Alice's public key, however, an unauthorized agent might intercept the key. Furthermore, the same agent might intercept the encrypted message from Bob. However, the agent cannot decrypt the message with the public key. The message can only be decrypted with Alice's private key, which has not been transmitted. Alice doesn't use her private key to encrypt a reply message to Bob, because anyone with the public key could decrypt the message. If Alice wants to send a message back to Bob, she asks Bob for his public key and encrypts her message using that public key. Bob then decrypts the message using his associated private key.

In a real world scenario, Alice and Bob use public key (asymmetric) encryption to transfer a secret (symmetric) key and use secret key encryption for the remainder of their session.

Public–key encryption has a much larger keyspace, or range of possible values for the key, and is therefore less susceptible to exhaustive attacks wherein every possibly key is tried. A public key is easy to distribute because it doesn't have to be secured. Public–key algorithms can be used to create digital signatures to verify the identity the sender of data. However, public–key algorithms are extremely slow (compared to secret key algorithms) and are not designed to encrypt large amounts of data. Public–key algorithms are useful only for transferring very small amounts of data. Typically, public–key encryption is used to encrypt a key and IV to be used by a secret–key algorithm. After the key and IV are transferred, then secret–key encryption is used for the remainder of the session.

The .NET Framework provides the following classes that implement public–key encryption algorithms:

- a.. DSACryptoServiceProvider
- b.. RSACryptoServiceProvider

Digital Signatures

Public–key algorithms can also be used to form digital signatures. Digital signatures authenticate the identity of a sender (if you trust the sender's public key) and protect the integrity of data. Using a public key generated by Alice, the recipient of Alice's data can verify that Alice sent it by comparing the digital signature to Alice's data and Alice's public key.

To use public–key cryptography to digitally sign a message, Alice first applies a hash algorithm to the message to create a message digest. The message digest is a compact and unique representation of data. Alice then encrypts the message digest with her private key to create her personal

signature. Upon receiving the message and signature, Bob decrypts the signature using Alice's public key to recover the message digest, and hashes the message using the same hash algorithm that Alice used. If the message digest that Bob computes exactly matches the message digest received from Alice, Bob is assured that the message came from the possessor of private key and that the data has not been modified. If Bob trusts that Alice is the possessor of the private key, then he knows that the message came from Alice.

Note that a signature can be verified by anyone because the sender's public key is common knowledge and is typically included in the digital signature format. This method does not retain the secrecy of the message; for the message to be secret, it must also be encrypted.

The .NET Framework provides the following classes that implement digital signature algorithms:

- a.. DSACryptoServiceProvider
- b.. RSACryptoServiceProvider

Hash Values

Hash algorithms map binary values of an arbitrary length to small binary values of a fixed length, known as hash values. A hash value is a unique and extremely compact numerical representation of a piece of data. If you hash a paragraph of plaintext and change even one letter of the paragraph, then a subsequent hash will produce a different value. It is computationally improbable to find two distinct inputs that hash to the same value.

Message authentication code (MAC) hash functions are commonly used with digital signatures to sign data while message detection code (MDC) hash functions are used for data integrity.

Alice and Bob might use a hash function in the following way to ensure data integrity. If Alice writes a message for Bob and creates a hash of that message, Bob can then hash the message at a later time and compare his hash to the original hash. If both hash values are identical, then the message was not altered; however, if the values are not identical, the message was altered after Alice wrote it. In order for this system to work, Alice must hide the original hash value from all parties except Bob.

The .NET Framework provides the following classes that implement digital signature algorithms:

- a.. HMACSHA1
- b.. MACTripleDES
- c.. MD5CryptoServiceProvider
- d.. SHA1Managed
- e.. SHA256Managed
- f.. SHA384Managed
- g.. SHA512Managed

Random Number Generation

Random number generation is integral to many cryptographic operations. For

example, cryptographic keys need to be as random as possible so that it is infeasible to reproduce them. Cryptographic random number generators must generate output that is computationally infeasible to predict with better than a probability of $p < .05$; that is, any method of predicting the next output bit must not perform better than random guessing. The classes in the .NET Framework use random number generators to generate cryptographic keys.

RNGCryptoServiceProvider is an implementation of a random number generator algorithm.

See Also

.NET Framework Cryptography Model | Cryptographic Tasks | Cryptographic Services

Syntax based on .NET Framework version 1.1.
Documentation version 1.1.0.

Send comments on this topic.
© 2001-2002 Microsoft Corporation. All rights reserved.
"Leonardo D'Ippolito" <leodippolito@terra.com.br> wrote in message
news:abe7f0f.0409112137.50aa58a2@posting.google.com...
> Hi!
>
> I have two .NET win apps that need to communicate on a TCP/IP network.
> 'App A' must ask 'app B' if it's allowed to do some task, and 'app B'
> must authorize or prohibit it.
>
> How can I do this kind of communication in a secure way (protected
> from sniffing)? It would be a very simple protocol. Question, and two
> possible answers 'yes' or 'no'.
>
> Thanks
>
> Leonardo