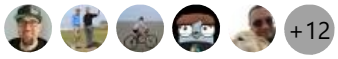
Cryptographic Services

07/14/2020 • 15 minutes to read • 

In this article

Public networks such as the Internet do not provide a means of secure communication between entities. Communication over such networks is susceptible to being read or even modified by unauthorized third parties. Cryptography helps protect data from being viewed, provides ways to detect whether data has been modified, and helps provide a secure means of communication over otherwise nonsecure channels. For example, data can be encrypted by 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.

In .NET, the classes in the System.Security.Cryptography namespace manage many details of cryptography for you. Some are wrappers for operating system implementations, while others are purely managed implementations. You do not need to be an expert in cryptography to use these classes. When you create a new instance of one of the encryption algorithm classes, keys are autogenerated for ease of use, and default properties are as safe and secure as possible.

This overview provides a synopsis of the encryption methods and practices supported by .NET, including the ClickOnce manifests.

Cryptographic Primitives

In a typical situation where cryptography is used, two parties (Alice and Bob) communicate over a nonsecure 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 make sure that the information she receives from Bob has not been modified by anyone during transmission. In addition, she must make sure that the information really does originate from Bob and not from someone who is impersonating Bob.

Cryptography is used to achieve the following goals:

- Confidentiality: To help protect a user's identity or data from being read.
- Data integrity: To help protect data from being changed.
- Authentication: To ensure that data originates from a particular party.
- Non-repudiation: To prevent a particular party from denying that they sent a message.

To achieve these goals, you can 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 to keep it 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 to keep it from being read by third parties. This type of encryption uses a public/private key pair to encrypt and decrypt data.
Cryptographic signing	Helps verify 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 your data or encrypt their own data, claiming it originated from you.

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 very fast (compared with public-key algorithms) and are well suited for performing cryptographic transformations on large streams of data. Asymmetric encryption algorithms such as RSA are limited mathematically in how much data they can encrypt. Symmetric encryption algorithms do not generally have those problems.

A type of secret-key algorithm called a block cipher is used to encrypt one block of data at a time. Block ciphers such as Data Encryption Standard (DES), TripleDES, and Advanced Encryption Standard (AES) 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 *n* is small (8 bytes for DES and TripleDES; 16 bytes [the default], 24 bytes, or 32 bytes for AES), data values that are larger than *n* have to be encrypted one block at a time. Data values that are smaller than *n* have to be expanded to *n* in order to be processed.

One simple form of block cipher is called the electronic codebook (ECB) mode. ECB mode is not considered secure, because it does not use an initialization vector to initialize the first plaintext block. 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. Therefore, if you have duplicate blocks in your input plaintext stream, you will have duplicate blocks in your output ciphertext stream. These duplicate output blocks alert unauthorized users to the weak encryption used the algorithms that might have been employed, and the possible modes of attack. The ECB cipher mode is therefore quite vulnerable to analysis, and ultimately, key discovery.

The block cipher classes that are provided in the base class library use a default chaining mode called cipher-block chaining (CBC), although you can change this default if you want.

CBC ciphers overcome the problems associated with ECB ciphers by using an initialization vector (IV) to encrypt the first block of plaintext. Each subsequent block of

plaintext undergoes a bitwise exclusive OR (`XOR`) operation with the previous ciphertext block before it is encrypted. Each ciphertext block is therefore dependent on all previous blocks. When this system is used, common message headers that might be known to an unauthorized user cannot be used to reverse-engineer a key.

One way to compromise data that is encrypted with a CBC cipher is to perform an exhaustive search of every possible key. Depending on the size of the key that is used to perform encryption, this kind of search is very time-consuming using even the fastest computers and is therefore infeasible. Larger key sizes are more difficult to decipher. Although encryption does not make it theoretically impossible for an adversary to retrieve the encrypted data, it does raise the cost of doing this. If it takes three months to perform an exhaustive search to retrieve data that is meaningful only for a few days, 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. The IV is not considered a secret and can be transmitted in plaintext with the message. However, the key must be kept secret from unauthorized users. Because of these problems, secret-key encryption is often used together 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 a nonsecure channel, they might use secret-key encryption as follows: Alice and Bob agree to use one particular algorithm (AES, for example) with a particular key and IV. Alice composes a message and creates a network stream (perhaps a named pipe or network email) on which to send the message. Next, she encrypts the text using the key and IV, and sends the encrypted message and IV to Bob over the intranet. Bob receives the encrypted text and decrypts it by using the IV and previously agreed upon key. If the transmission is intercepted, the interceptor cannot recover the original message, because they do not know the key. In this scenario, only the key must remain secret. In a real world scenario, either Alice or Bob generates a secret key and uses public-key (asymmetric) encryption to transfer the secret (symmetric) key to the other party. For more information about public-key encryption, see the next section.

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

- Aes
- HMACSHA256, HMACSHA384 and HMACSHA512. (These are technically secret-key

algorithms because they represent message authentication codes that are calculated by using a cryptographic hash function combined with a secret key. See Hash Values, later in this article.)

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. The public key and the private key are mathematically linked; data that is encrypted with the public key can be decrypted only with the private key, and data that is signed with the private key can be verified only with the public key. The public key can be made available to anyone; it is used for encrypting data to be sent to the keeper of the private key. Public-key cryptographic algorithms are also known as asymmetric algorithms because one key is required to encrypt data, and another key is required to decrypt data. A basic cryptographic rule prohibits key reuse, and both keys should be unique for each communication session. However, in practice, asymmetric keys are generally long-lived.

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 a nonsecure network, and Bob uses this key to encrypt a message. Bob sends the encrypted message to Alice, and she decrypts it by using her private key. If Bob received Alice's key over a nonsecure channel, such as a public network, Bob is open to a man-in-the-middle attack. Therefore, Bob must verify with Alice that he has a correct copy of her public key.

During the transmission of Alice's public key, 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 be decrypted only with Alice's private key, which has not been transmitted. Alice does not 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 this 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.

The following list offers comparisons between public-key and secret-key cryptographic

algorithms:

- 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.
- Public-key encryption has a much larger keyspace (range of possible values for the key) than secret-key encryption. Therefore, public-key encryption is less susceptible to exhaustive attacks that try every possible key.
- Public keys are easy to distribute because they do not have to be secured, provided that some way exists to verify the identity of the sender.
- Some public-key algorithms (such as RSA and DSA, but not Diffie-Hellman) can be used to create digital signatures to verify the identity of the sender of data.
- Public-key algorithms are very slow compared with 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, secret-key encryption is used for the remainder of the session.

.NET provides the following classes that implement public-key algorithms:

- RSA
- ECDsa
- ECDiffieHellman
- DSA

RSA allows both encryption and signing, but DSA can be used only for signing. DSA is not as secure as RSA, and we recommend RSA. Diffie-Hellman can be used only for key generation. In general, public-key algorithms are more limited in their uses than private-key algorithms.

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 help 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 the private key and that the data has not been modified. If Bob trusts that Alice is the possessor of the private key, he knows that the message came from Alice.

> **Note**
>
> 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.

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

- RSA
- ECDsa
- DSA

Hash Values

Hash algorithms map binary values of an arbitrary length to smaller binary values of a fixed length, known as hash values. A hash value is a numerical representation of a piece

of data. If you hash a paragraph of plaintext and change even one letter of the paragraph, a subsequent hash will produce a different value. If the hash is cryptographically strong, its value will change significantly. For example, if a single bit of a message is changed, a strong hash function may produce an output that differs by 50 percent. Many input values may hash to the same output value. However, it is computationally infeasible to find two distinct inputs that hash to the same value.

Two parties (Alice and Bob) could use a hash function to ensure message integrity. They would select a hash algorithm to sign their messages. Alice would write a message, and then create a hash of that message by using the selected algorithm. They would then follow one of the following methods:

- Alice sends the plaintext message and the hashed message (digital signature) to Bob. Bob receives and hashes the message and compares his hash value to the hash value that he received from Alice. If the hash values are identical, the message was not altered. If the values are not identical, the message was altered after Alice wrote it.

  Unfortunately, this method does not establish the authenticity of the sender. Anyone can impersonate Alice and send a message to Bob. They can use the same hash algorithm to sign their message, and all Bob can determine is that the message matches its signature. This is one form of a man-in-the-middle attack. For more information, see Cryptography Next Generation (CNG) Secure Communication Example.

- Alice sends the plaintext message to Bob over a nonsecure public channel. She sends the hashed message to Bob over a secure private channel. Bob receives the plaintext message, hashes it, and compares the hash to the privately exchanged hash. If the hashes match, Bob knows two things:

  - The message was not altered.
  - The sender of the message (Alice) is authentic.

  For this system to work, Alice must hide her original hash value from all parties except Bob.

- Alice sends the plaintext message to Bob over a nonsecure public channel and places the hashed message on her publicly viewable Web site.

This method prevents message tampering by preventing anyone from modifying the hash value. Although the message and its hash can be read by anyone, the hash value can be changed only by Alice. An attacker who wants to impersonate Alice would require access to Alice's Web site.

None of the previous methods will prevent someone from reading Alice's messages, because they are transmitted in plaintext. Full security typically requires digital signatures (message signing) and encryption.

.NET provides the following classes that implement hashing algorithms:

- SHA256.
- SHA384.
- SHA512.

.NET also provides MD5 and SHA1. But the MD5 and SHA-1 algorithms have been found to be insecure, and SHA-2 is now recommended instead. SHA-2 includes SHA256, SHA384, and SHA512.

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 a probability that is better than one half. Therefore, any method of predicting the next output bit must not perform better than random guessing. The classes in .NET use random number generators to generate cryptographic keys.

The RandomNumberGenerator class is an implementation of a random number generator algorithm.

ClickOnce Manifests

The following cryptography classes let you obtain and verify information about manifest signatures for applications that are deployed using ClickOnce technology:

- The ManifestSignatureInformation class obtains information about a manifest signature when you use its VerifySignature method overloads.
- You can use the ManifestKinds enumeration to specify which manifests to verify. The result of the verification is one of the SignatureVerificationResult enumeration values.
- The ManifestSignatureInformationCollection class provides a read-only collection of ManifestSignatureInformation objects of the verified signatures.

In addition, the following classes provide specific signature information:

- StrongNameSignatureInformation holds the strong name signature information for a manifest.
- AuthenticodeSignatureInformation represents the Authenticode signature information for a manifest.
- TimestampInformation contains information about the time stamp on an Authenticode signature.
- TrustStatus provides a simple way to check whether an Authenticode signature is trusted.

Cryptography Next Generation (CNG) Classes

The Cryptography Next Generation (CNG) classes provide a managed wrapper around the native CNG functions. (CNG is the replacement for CryptoAPI.) These classes have "Cng" as part of their names. Central to the CNG wrapper classes is the CngKey key container class, which abstracts the storage and use of CNG keys. This class lets you store a key pair or a public key securely and refer to it by using a simple string name. The elliptic curve-based ECDsaCng signature class and the ECDiffieHellmanCng encryption class can use CngKey objects.

The CngKey class is used for a variety of additional operations, including opening, creating, deleting, and exporting keys. It also provides access to the underlying key handle to use when calling native functions directly.

.NET also includes a variety of supporting CNG classes, such as the following:

- CngProvider maintains a key storage provider.
- CngAlgorithm maintains a CNG algorithm.
- CngProperty maintains frequently used key properties.

See also

- Cryptography Model - Describes how cryptography is implemented in the base class library.
- Cross-Platform Cryptography
- Timing vulnerabilities with CBC-mode symmetric decryption using padding
- ASP.NET Core Data Protection

Is this page helpful?

Yes No

Recommended content

How to: store asymmetric keys in a key container

Learn how to store asymmetric keys in a key container in .NET. See how to create an asymmetric key, save it in a key container, and retrieve and delete the key.

CipherMode Enum (System.Security.Cryptography)

Specifies the block cipher mode to use for encryption.

Cryptographic Signatures

Learn more about: Cryptographic Signatures

RSA Class (System.Security.Cryptography)

Represents the base class from which all implementations of the RSA algorithm inherit.

RSACryptoServiceProvider Class (System.Security.Cryptography)

Performs asymmetric encryption and decryption using the implementation of the RSA algorithm provided by the cryptographic service provider (CSP). This class cannot be inherited.

AesCng Class (System.Security.Cryptography)

Provides a Cryptography Next Generation (CNG) implementation of the Advanced Encryption Standard (AES) algorithm.

AesCryptoServiceProvider Class (System.Security.Cryptography)

Performs symmetric encryption and decryption using the Cryptographic Application Programming Interfaces (CAPI) implementation of the Advanced Encryption Standard (AES) algorithm.

RSA.ExportRSAPublicKey Method (System.Security.Cryptography)

Exports the public-key portion of the current key in the PKCS#1 RSAPublicKey format.

Show less