Capítulo 8
Criptografia

Introdução

Criptografia de dados é um ponto muito importante nos mais diversos tipos de aplicações. Geralmente, em aplicações onde alguns dos dados são muito sigilosos, como é o caso de aplicações financeiras, quais mantém os dados de seus clientes, é necessário que se mantenha esses dados seguros e desejavelmente, se esses dados cairem em mãos erradas, essas pessoas com más intenções, não consigam entender e recuperar esses dados em sua forma legível.

É justamente neste cenário que a criptografia entra, ou seja, ao se detectar os dados sensíveis durante a análise do projeto, deve ser aplicar algum algoritmo de criptografia para manter esses dados o mais seguro possível. Isso é vital para uma aplicação deste nível.

Neste capítulo analisaremos as possibilidades que o .NET Framework nos fornece para aplicar criptografia em dados, bem como executar o processo reverso, ou seja, ser capaz de trazer os dados em sua forma legível. Todos os recursos estão contidos dentro do *namespace* **System.Security.Cryptography**. Ainda há a possibilidade de efetuarmos a criptografia em uma única direção, ou seja, não sermos mais capazes de recuperar o seu conteúdo em sua forma legível. Esse processo é conhecido como *hash* e será extensivamente abordado na segunda parte deste capítulo.

O que é criptografia?

Criptografia trata-se de o processo de converter alguns dados em um formato difícil de ler e compreender. O processo de criptografia é capaz de, dado um conteúdo qualquer (número, letras, etc.), transformá-lo em uma seqüencia de caracteres que, a olha humano, não é capaz de ser traduzido. A criptografia é utilizada para:

1. **Confidencialidade:** Para garantir que os dados permaneçam privados. Geralmente, a confidencialidade é obtida com a criptografia. Os algoritmos de criptografia (que usam chaves de criptografia) são usados para converter texto sem formatação em texto codificado e o algoritmo de descriptografia equivalente é usado para converter o texto codificado em texto sem formatação novamente. Os algoritmos de criptografia simétricos usam a mesma chave para a criptografia e a descriptografia, enquanto que os algoritmos assimétricos usam um par de chaves pública/privada.
2. **Integridade de dados:** Para garantir que os dados sejam protegidos contra modificação acidental ou deliberada (mal-intencionada). A integridade, geralmente, é fornecida por códigos de autenticação de mensagem ou *hashes*. Um valor de *hash* é um valor numérico de comprimento fixo derivado de uma seqüência de dados. Os valores de *hash* são usados para verificar a integridade

dos dados enviados por canais não seguros. O valor do *hash* de dados recebidos é comparado ao valor do *hash* dos dados, conforme eles foram enviados para determinar se foram alterados.

3. **Autenticação:** Para garantir que os dados se originem de uma parte específica. Os certificados digitais são usados para fornecer autenticação. As assinaturas digitais geralmente são aplicadas a valores de *hash*, uma vez que eles são significativamente menores que os dados de origem que representam.

Os algoritmos de criptografia utilizam um valor um tanto quanto complexo, chamado de *cipher* (ou como *key*), para ser utilizado no processo de criptografia. O processo que utiliza o *cipher* para encriptar os dados e produzir o valor já criptografado é chamado de *algoritmo de criptografia*. Os valores que compõem os *cipher* podem ser de diferentes tamanhos e, quanto maior, mais seguro é e, conseqüentemente, mais difícil de ser "quebrado".

Como já dissemos na introdução do capítulo, o processo de criptografia é um processo que consiste em duas rotinas: criptografar os dados e decriptografá-los, que é justamente o processo inverso, ou seja, dado o valor criptografado, ele é capaz de recuperar o seu conteúdo em sua legível. Mas é importante lembrar que esse processo somente é possível se a chave utilizada for a mesma chave (*cipher*) aplicada durante o processo de criptografia.

Dentro do processo de criptografia, existem duas categorias de algoritmos. Essas categorias são baseadas em que *cipher* é utilizado e como ele é gerenciado. Essas categorias são chamadas de **criptografia simétrica** e **criptografia assimétrica**.

A primeira delas, a criptografia simétrica, utiliza uma chave única privada, tanto para criptografar quanto para descriptografar os dados. Já a criptografia assimétrica, utiliza um par de chaves, sendo uma pública e uma privada. Uma delas é mantida privada e a outra é ditribuída publicamente.

Criptografia simétrica

A criptografia simétrica utiliza apenas uma chave privada que é necessária tanto para criptografar quanto para descriptografar as informações e é utilizada em um grupo pequenos de parceiros e companhias.

Os algoritmos de criptografia simétricas são menos complexos e mais eficiente que os algoritmos de criprografia assimétricas, justamente porque possuem apenas uma única chave. Sendo assim, se essa chave cair nas mãos de pessoas erradas, poderá comprometer a segurança dos dados, já que ela poderá descriptografar os dados sem nenhum outro problema e, se isso realmente acontecer, você deve obrigatoriamente e o mais rápido possível trocar essa chave. Devido a essa enorme responsabilidade de ter que manter essa chave segura, muitas empresas optam por utilizar a criptografia assimétrica, que veremos mais adiante.

Dentro do .NET Framework temos os mais conhecidos algoritmos de criptografia simétrica implementadas em diversas classes. Essas classes estão contidas dentro do *namespace* **System.Security.Cryptography**. Entre os algoritmos temos: **Data Encryption Standard (DES)**, **Triple DES**, **RC2 (Rivest Cipher)**, e **Rijndael**. O imagem abaixo ilustra a hierarquia das classes que fornecem os mais algoritmos simétricos:

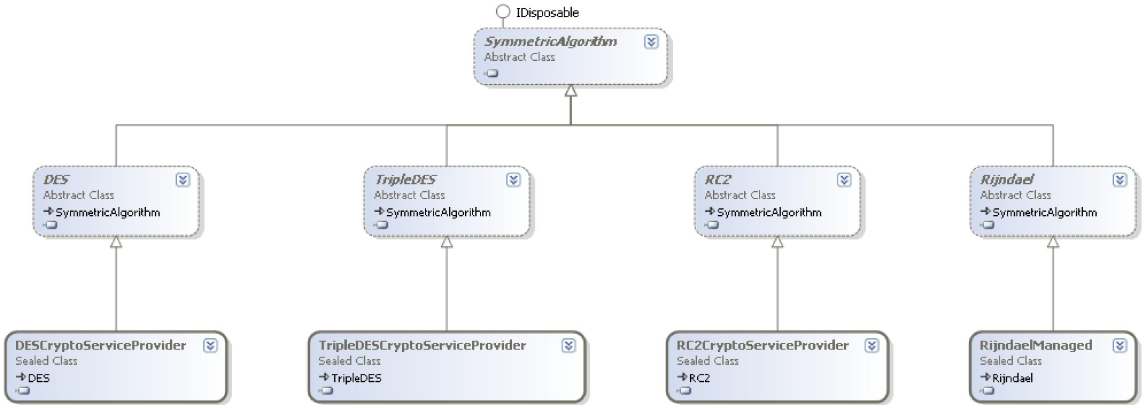

Imagem 8.1 – Hierarquia das classes de criptografia simétrica

Como podemos notar, todas as classes desta categoria de criptografia herdam diretamente da classe abstrata **SymmetricAlgorithm**, que fornece todas as funcionalidades básicas para qualquer algoritmo de criptografia simétrica. Todas as classes que herdam dela, como é o caso das classes **DES**, **TripleDES**, **RC2** e **Rijndael**, são também definidas como abstratas e, mais tarde, serão implementadas pela classe concreta, que veremos a seguir.

Cada uma dessas classes que herdam diretamente da classe abstrata **SymmetricAlgorithm**, são agora herdadas, criando um **CSP** (*Cryptographic Service Provider*) correspondente para cada um dos algoritmos. Um **CSP** é um módulo independente que atualmente executa os algoritmos de criptografia para autenticação, codificação e criptografia, ou seja, é basicamente um *wrapper* para o algoritmo de criptografia correspondente, encapsulando o acesso a objetos não gerenciados pelo *runtime* do .NET Framework que são utilizados durante o processo de criptografia.

Para termos uma noção melhor do que cada uma das classes fornecem, vamos analisar a relação abaixo:

Algoritmo	Descrição
DES	Criado pela IBM em 1975, suporta uma chave de 56 *bits*, mas dentro do .NET Framework, possibilita o uso de uma chave de 64 *bits*. O .NET Framework suporta apenas porque esse algoritmo é popularmente utilizado, mas já foi "quebrado". Essa classe é base para todos os algoritmos baseando em **DES** e sua *CSP* correspondente é a classe

	DESCryptoServiceProvider. Essa classe ainda possui dois métodos públicos e estáticos importantes, que analisam a complexidade e determinam se a chave é ou não fácil de ser "quebrada". Esses métodos são invocados durante o proceso de criptografia e descriptografia e atira uma exceção do tipo **CryptographicException** caso a chave seja fácil de ser "quebrada", o que obriga-nos a chamar os métodos dentro de um bloco de tratamento de erros para antecipar esses possíveis problemas. Os métodos são: • **IsWeakKey:** Existem quatro chaves que são consideradas fracas e facilmente de serem quebradas. Esse método retorna um valor booleano indicando se a chave fornecida é ou não fraca. • **IsSemiWeakKey:** Existe seis chaves que são consideradas semi-fracas e podem ser facilmente quebradas. Esse método retorna um valor booleano indicando se a chave fornecida é ou não semi-fraca. **Nota:** Felizmente o método *GenerateKey* nunca retornará uma chave que é fraca ou semi-fraca.
TripleDES	Criado pela IBM em 1978, suporta uma chave de 168 *bits*, mas dentro do .NET Framework, possibilita o uso de uma chave de 128 até 192 *bits*. Essa classe é base para todos os algoritmos baseando em **TripleDES** e sua *CSP* correspondente é a classe **TripleDESCryptoServiceProvider**. Esse algoritmo é baseado no algoritmo **DES** e, sendo assim, contém também chaves conhecidas que podem quebrar a criptografia. No entanto, esse algoritmo é considerado muito mais seguro em comparação ao **DES**, justamente por ter uma chave maior e o algoritmo é aplicado três vezes antes de disponibilizar o resultado. Essa classe também possui um método público e estático chamado *IsWeakKey* que retorna um valor booleano indicando se a chave fornecida é ou não fraca.
RC2	Representa a classe base para todas as implementações do algoritmo RC2. Esse algoritmo foi criado em 1987 por Ron Rivest. Esse algoritmo suporta chaves de 40 até 128 *bits*. Essa classe é base para todos os algoritmos baseando em **RC2** e sua *CSP* correspondente é a classe **RC2CryptoServiceProvider**.
Rijndael	Criado em 1998 por Joan Daemen e Vincent Rijmen, Rijndael também é conhecida como **Advanced Encryption Standard (AES)**. Essa classe suporta chaves de 128 até 256 *bits* e, dentro do .NET Framework, o algoritmo **Rijndael** suporta valores fixos de 128, 192 ou 256 *bits*. Essa classe é base para todos os algoritmos baseando em **Rijndael** e a classe concreta que implementa essa algoritmo dentro do .NET Framework é a

	classe **RijndaelManaged**.

Tipos adicionais

Antes de analisarmos exemplos de códigos que utilizamos para efetuarmos os processos de criptografia e descriptografia, devemos estudar um pouco mais sobre alguns tipos adicionais que temos dentro do .NET Framework, que são utilizados em conjunto com as classes responsáveis pela criptografia em si.

O primeiro deles é a classe **CryptoStream**. Essa classe herda diretamente da classe **Stream** e fornece um link entre o *stream* de dados e as transformações realizadas durante o processo de criptografia. Seu construtor recebe três parâmetros: um objeto do tipo **Stream**, que é onde os dados criptografados serão armazenados; já o segundo parâmetro, recebe um objeto do tipo **ICryptoTransform**. Essa *Interface* é implementada em classes que define as operações básicas de transformações criptográficas. Instâncias de objetos que implementam esta *Interface* são retornados quando invocamos os métodos *CreateEncryptor* e *CreateDecryptor* referente à algum algoritmo fornecido pela plataforma .NET. Finalmente, o terceiro e último parâmetro, recebe uma das opções fornecidas pelo enumerador **CryptoStreamMode**, que irá indicar qual será o modo que o *stream* será operado. As opções que temos neste enumerador são:

Opção	Descrição
Read	Permite acesso à leitura ao *stream* de criptografia.
Write	Permite acesso à escrita ao *stream* de criptografia.

Além da classe **CryptoStream** ainda fazemos uso de algumas classes contidas dentro do *namespace* **System.IO** para poder efetuar a leitura e a escrita dos dados criptografados. Entre essas classes temos a classe **MemoryStream, StreamWriter** e **StreamReader**, quais já abordamos anteriormente no *Capítulo 5*. No cenário de criptografia, a classe **MemoryStream** é utilizada para armazenar os dos que serão criptografados ou descriptografados; já as classes **StreamWriter** e **StreamReader** são utilizadas no processo de criptografia e descriptografia de dados, respectivamente.

Por questões de espaço, não abordaremos todos os algoritmos de criptografia simétrica aqui. Apenas como exemplo, teremos a implementação aqui do algoritmo **DES** e na aplicação de demonstração do capítulo teremos todos os algoritmos implementados. Sendo assim, o exemplo abaixo utiliza um único *CSP* para efetuar tanto a criptografia quanto a descriptografia de uma mensagem simples.

```
VB.NET
Imports System
Imports System.IO
Imports System.Text
Imports System.Security.Cryptography
```

```
Using csp As New DESCryptoServiceProvider()
    Dim buffer() As Byte = Nothing

    Using ms As New MemoryStream()
        Using stream As New CryptoStream(ms, csp.CreateEncryptor,
CryptoStreamMode.Write)
            Using sw As New StreamWriter(stream)
                sw.Write("Trabalhando com Criptografia")
            End Using
        End Using

        buffer = ms.ToArray()
        Console.WriteLine(Encoding.Default.GetString(buffer))
    End Using

    Using ms As New MemoryStream(buffer)
        Using stream As New CryptoStream(ms,
csp.CreateDecryptor(), CryptoStreamMode.Read)
            Using sr As New StreamReader(stream)
                Console.WriteLine(sr.ReadLine())
            End Using
        End Using
    End Using
End Using
```

C#

```
using System;
using System.IO;
using System.Text;
using System.Security.Cryptography;

using (DESCryptoServiceProvider csp =
    new DESCryptoServiceProvider())
{
    byte[] buffer = null;

    using (MemoryStream ms = new MemoryStream())
    {
        using (CryptoStream stream =
            new CryptoStream(ms, csp.CreateEncryptor(),
CryptoStreamMode.Write))
        {
            using (StreamWriter sw = new StreamWriter(stream))
            {
                sw.WriteLine("Trabalhando com Criptografia");
            }
        }

        buffer = ms.ToArray();
        Console.WriteLine(Encoding.Default.GetString(buffer));
```

```
    }

    using (MemoryStream ms = new MemoryStream(buffer))
    {
        using (CryptoStream stream =
            new      CryptoStream(ms,      csp.CreateDecryptor(),
CryptoStreamMode.Read))
        {
            using (StreamReader sr = new StreamReader(stream))
            {
                Console.WriteLine(sr.ReadLine());
            }
        }
    }
}
```

Analisando o código acima e como já foi dito, utilizamos um único *CSP* do tipo **DESCryptoServiceProvider** que é utilizado tanto para criptografar quanto para descriptografar a mensagem. O método *CreateEncryptor* e *CreateDecryptor* são sobrecarregados; para cada um deles existe a possibilidade de invocá-los sem a necessidade de parâmetros e, neste caso, os valores da chave (propriedade *Key*) e do vetor (propriedade *IV*) são passados automaticamente e, se esses valores não foram definidos pelo desenvolvedor, como foi o caso acima, os métodos *GenerateKey* e *GenerateIV* são invocados.

Nota: IV (Initialization Vector) trata-se de uma valor que é aplicado na criptografia simétrica para garantir que uma mesma mensagem não seja criptografada da mesma forma, o que poderia corromper o algoritmo, ou seja, sabendo que a palavra "Microsoft" fosse sempre criptografada com um determinado conjunto de caracteres, bastaria percorrer o restante da mensagem e onde fosse encontrado esse conjunto, sabe-se que é a palavra "Microsoft" que ali está.

Como estamos fazendo os dois processos de uma única vez, não precisamos nos preocupar com a chave e o vetor que são automaticamente gerados, justamente porque utilizamos a mesma instância do *CSP* **DESCryptoServiceProvider**. Obviamente que esses valores devem ser guardados com muita segurança se futuramente você desejar descriptografar a mensagem. Lembrando que esses valores também devem ser distribuídos para todos que podem descriptografar os dados.

Criptografia Assimétrica

Também conhecida como "chave pública", a chave assimétrica trabalha com duas chaves: uma denominada privada e a outra pública. Nesse método, uma pessoa deve criar uma chave de codificação e enviá-la a quem for mandar informações a ela. Essa é a chave pública. Uma outra chave deve ser criada para a decodificação. Esta chave, também conhecida como chave privada, é secreta e não deve ser compartilhada.

Criptografia assimétrica é utilizada em larga escala, pois é ideal para cenários como por exemplo a própria Internet, onde a chave privada permanece secreta e a chave pública será largamente distribuída. Um outro exemplo típico de criptografia assimétrica é utilizada em assinaturas digitais, que é muito usado com chaves públicas. Trata-se de um meio que permite provar que um determinado documento eletrônico é de procedência verdadeira. O receptor da informação usará a chave pública fornecida pelo emissor para se certificar da origem. Além disso, a chave fica integrada ao documento de forma que qualquer alteração por terceiros imediatamente a invalide. O .NET Framework utiliza essa técnica quando "assinamos" um *Assembly* com um *strong name*.

Como algoritmos de criptografia assimétrica são considerados mais complexos em relação aos algoritmos de criptografia simétrica, a criptografia assimétrica será executada com mais lentidão.

Dentro do .NET Framework temos os mais conhecidos algoritmos de criptografia assimétrica implementadas em diversas classes. Essas classes estão contidas dentro do *namespace* **System.Security.Cryptography**. Entre os algoritmos temos o **DSA** e o **RSA**. O imagem abaixo ilustra a hierarquia das classes que fornecem os mais algoritmos assimétricos:

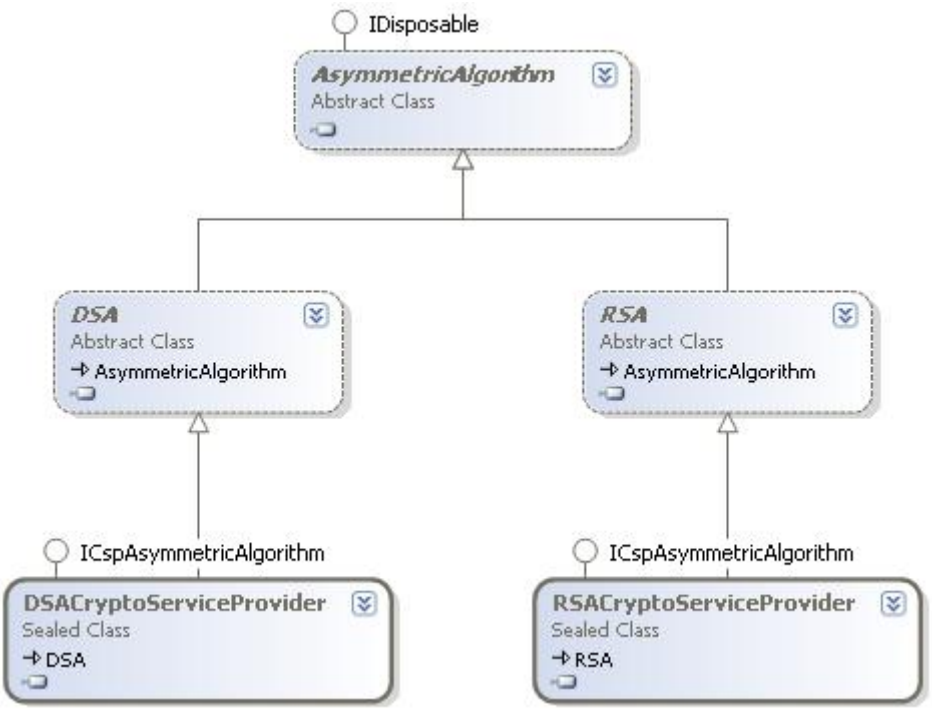

Imagem 8.2 – Hierarquia das classes de criptografia assimétrica

Como podemos notar, todas as classes desta categoria de criptografia herdam diretamente da classe abstrata **AsymmetricAlgorithm**, que fornece todas as funcionalidades básicas para qualquer algoritmo de criptografia simétrica. Todas as classes que herdam dela,

como é o caso das classes **DSA** e **RSA**, são também definidas como abstratas e, mais tarde, serão implementadas pela classe concreta, que veremos a seguir.

Cada uma dessas classes que herdam diretamente da classe abstrata **AsymmetricAlgorithm**, são agora herdadas, criando um **CSP** (*Cryptographic Service Provider*) correspondente para cada um dos algoritmos. Um **CSP** é um módulo independente que atualmente executa os algoritmos de criptografia para autenticação, codificação e criptografia, ou seja, é basicamente um *wrapper* para o algoritmo de criptografia correspondente.

Para termos uma noção melhor do que cada uma das classes fornecem, vamos analisar a relação abaixo:

Algoritmo	**Descrição**
DSA	Criado em 1991, o **DSA (Digital Signature Algorithm)** utiliza uma chave de 512 até 1024 *bits* e é utilizada como base para todos os algoritmos de assinatura digital. Essa classe é base para todos os algoritmos baseando em **DSA** e sua *CSP* correspondente é a classe **DSACryptoServiceProvider**.
RSA	Criado em 1977 por **R**on Rivest, Adi **S**hamir e Len **A**dleman é um dos algoritmos mais utilizados atualmente. Dentro do .NET Framework é suportado uma chave de 384 até 16.384 *bits* se utilizar o *Microsoft Enhanced Cryptographic Provider* e uma chave de 384 até 512 *bits* se utilizar o *Microsoft Base Cryptographic Provider*. Essa classe é base para todos os algoritmos baseando em **RSA** e sua *CSP* correspondente é a classe **RSACryptoServiceProvider**.

DSA

Como já vimos, o **DSA** é um algoritmo assimétrico e a sua chave privada opera sobre o *hash* da mensagem *SHA-1*. Para verificar a assinatura um pedaço do código calcula o *hash* e outro pedaço usa a chave pública para decifrar a assinatura, e por fim ambos comparam os resultados garantindo a autoria da mensagem.

O **DSA** trabalha com chaves de 512 até 1024 *bits*, porém ao contrário do **RSA**, o **DSA** somente assina e não garante confidencialidade. Outro ponto contra o **DSA** é que a geração da assinatura é mais rápida do que o **RSA**, porém de 10 até 40 vezes mais lenta para conferir a assinatura. Para exemplificar a utilização deste algoritmo, iremos utilizar a classe **DSACryptoServiceProvider** fornecida pelo .NET Framework. Além dessa classe, ainda temos outras duas classes que devemos utilizar: **DSASignatureFormatter** e **DSASignatureDeformatter**. A primeira delas é utilizada para criar um algoritmo de assinatura digital através de um método chamado *CreateSignature*; já a segunda, é utilizada para verificar se uma determinada assinatura é ou não válida, também através de um método chamado *VerifySignature*. O código mostra como fazermos para utilizar as classes acima mencionadas:

VB.NET

```
Imports System
Imports System.Security.Cryptography

Dim hash As Byte() = {22, 45, 78, 53, 1, 2, 205, 98, 75, 123, 45,
76, 143, 189, 205, 65, 12, 193, 211, 255}
Dim algoritmo As String = "SHA1"

Using csp As New DSACryptoServiceProvider
    Dim formatter As New DSASignatureFormatter(csp)
    formatter.SetHashAlgorithm(algoritmo)
    Dim signedHash As Byte() = formatter.CreateSignature(hash)

    Dim deformatter As New DSASignatureDeformatter(csp)
    deformatter.SetHashAlgorithm(algoritmo)

    If deformatter.VerifySignature(hash, signedHash) Then
        Console.WriteLine("Assinatura válida.")
    Else
        Console.WriteLine("Assinatura inválida.")
    End If
End Using
```

C#

```
using System;
using System.Security.Cryptography;

byte[] hash = { 22, 45, 78, 53, 1, 2, 205, 98, 75, 123, 45, 76,
143, 189, 205, 65, 12, 193, 211, 255 };
string algoritmo = "SHA1";

using       (DSACryptoServiceProvider       csp       =       new
DSACryptoServiceProvider())
{
    DSASignatureFormatter         formatter         =         new
DSASignatureFormatter(csp);
    formatter.SetHashAlgorithm(algoritmo);
    byte[] signedHash = formatter.CreateSignature(hash);

    DSASignatureDeformatter        deformatter        =        new
DSASignatureDeformatter(csp);
    deformatter.SetHashAlgorithm(algoritmo);

    if (deformatter.VerifySignature(hash, signedHash))
    {
        Console.WriteLine("Assinatura válida.");
    }
    else
```

```
    {
        Console.WriteLine("Assinatura inválida.");
    }
}
```

RSA

O exemplo do algoritmo **RSA** é mais simples em relação ao algoritmo **DSA**. Dentro do .NET Framework temos a classe (*CSP*) **RSACryptoServiceProvider** que fornece uma implementação do algoritmo **RSA**. Essa classe fornece dois métodos chamados *Encrypt* e *Decrypt*. Ambos recebem um *array* de *bytes* onde, no primeiro caso, o método *Encrypt*, o array de bytes representa a mensagem a ser criptografada e retornará um *array* de *bytes* com a mensagem criptografada; já o segundo método, *Decrypt*, receberá um *array* de *bytes* contendo a mensagem criptografada e retornará um *array* de *bytes* com a mensagem descriptografada.

O exemplo abaixo exibe a utilização da implementação do algoritmo **RSA** fornecido pelo .NET Framework:

VB.NET

```
Imports System
Imports System.Security.Cryptography

Using csp As New RSACryptoServiceProvider
    Dim msg As Byte() = Encoding.Default.GetBytes("Trabalhando
com criptografia")
    Dim msgEncriptada As Byte() = csp.Encrypt(msg, True)
    Console.WriteLine(Encoding.Default.GetString(msgEncriptada))

    Dim msgDescriptada As Byte() = csp.Decrypt(msgEncriptada,
True)
    Console.WriteLine(Encoding.Default.GetString(msgDescriptada))
End Using
```

C#

```
using System;
using System.Security.Cryptography;

using (RSACryptoServiceProvider csp = new
RSACryptoServiceProvider())
{
    byte[] msg = Encoding.Default.GetBytes("Trabalhando com
criptografia");
    byte[] msgEncriptada = csp.Encrypt(msg, true);
    Console.WriteLine(Encoding.Default.GetString(msgEncriptada));
```

```
    byte[] msgDescriptada = csp.Decrypt(msgEncriptada, true);

Console.WriteLine(Encoding.Default.GetString(msgDescriptada));
}
```

CryptoConfig

Essa classe é utilizada em aplicações que não podem (ou não querem) depender de um algoritmo específico de criptografia ou *hashing*. Isso pode aumentar a segurança, já que a aplicação poderá ser flexível ao ponto de escolher um entre uma porção de algoritmos disponíveis e aplicar a criptografia/*hashing*. Pode-se optar por armazenar o algoritmo utilizado dentro da base de dados e, a qualquer momento que desejar, pode recuperar tanto o valor criptografado quanto o algoritmo que deve ser aplicado para descriptografar o valor.

Essa classe possui um método chamado estático *CreateFromName* que recebe como parâmetro uma *string* contendo o algoritmo concreto que deseja criar, retornando um **System.Object** contendo a classe devidamente instanciada. Esse método ainda possui um *overload* que recebe um *array* de **System.Object** que são os parâmetros que devem ser passados para o construtor da classe, quando houver. A *string* que será informada para o método deve estar dentro de uma das opções da tabela abaixo:

Nome/Chave	**Algoritmo que será instanciado**
SHA	SHA1CryptoServiceProvider
SHA1	SHA1CryptoServiceProvider
System.Security.Cryptography.SHA1	SHA1CryptoServiceProvider
System.Security.Cryptography.HashAlgorithm	SHA1CryptoServiceProvider
MD5	MD5CryptoServiceProvider
System.Security.Cryptography.MD5	MD5CryptoServiceProvider
SHA256	SHA256Managed
SHA-256	SHA256Managed
System.Security.Cryptography.SHA256	SHA256Managed
SHA384	SHA384Managed
SHA-384	SHA384Managed
System.Security.Cryptography.SHA384	SHA384Managed
SHA512	SHA512Managed
SHA-512	SHA512Managed
System.Security.Cryptography.SHA512	SHA512Managed
RSA	RSACryptoServiceProvider
System.Security.Cryptography.RSA	RSACryptoServiceProvider
System.Security.Cryptography.AsymmetricAlgorithm	RSACryptoServiceProvider
DSA	DSACryptoServiceProvider
System.Security.Cryptography.DSA	DSACryptoServiceProvider

DES	DESCryptoServiceProvider
System.Security.Cryptography.DES	DESCryptoServiceProvider
3DES	TripleDESCryptoServiceProvider
TripleDES	TripleDESCryptoServiceProvider
Triple DES	TripleDESCryptoServiceProvider
System.Security.Cryptography.TripleDES	TripleDESCryptoServiceProvider
System.Security.Cryptography.SymmetricAlgorithm	TripleDESCryptoServiceProvider
RC2	RC2CryptoServiceProvider
System.Security.Cryptography.RC2	RC2CryptoServiceProvider
Rijndael	RijndaelManaged
System.Security.Cryptography.Rijndael	RijndaelManaged

O código abaixo exibe uma forma de utilizar essa classe, passando como parâmetro para o método *CreateFromName* o valor "SHA", que fará com que uma instância do algoritmo do tipo **SHA1CryptoServiceProvider** seja retornado.

```
VB.NET
Imports System
Imports System.Security.Cryptography

Dim csp As SHA1CryptoServiceProvider =
    DirectCast(CryptoConfig.CreateFromName("SHA"),
SHA1CryptoServiceProvider)

C#
using System;
using System.Security.Cryptography;

SHA1CryptoServiceProvider csp =

(SHA1CryptoServiceProvider)CryptoConfig.CreateFromName("SHA");
```

O que é hashing?

Ao contrário dos tipos de criptografias que vimos até o momento, o objetivo do *hash* não é garantir a confidenciabilidade de uma determinada informação, mas sim garantir que a mesma não sofra nenhuma espécie de adulteração.

Sendo assim, a técnica de hashing é considera unidirecional, ou seja, uma vez aplicado o algoritmo de *hashing* em uma informação, jamais será possível recuperar aquela informação em seu estado legível. Isso quer dizer que um determinado valor sempre gerará o mesmo código *hash* e, para comparar se os dois valores *hashes* são iguais, você deve aplicar o algoritmo *hash* ao valor informado pelo usuário e, o valor gerado, deve ser comparado quanto à igualdade ao valor *hash* que tem armazenado dentro do sistema.

Obviamente que o mesmo algoritmo deve ser aplicado para garantir que seja possível efetuar a comparação.

O .NET Framework suporte três algoritmos de *hash*, a saber: **MD5**, **SHA1**e **HMAC** e estão assim distribuídas:

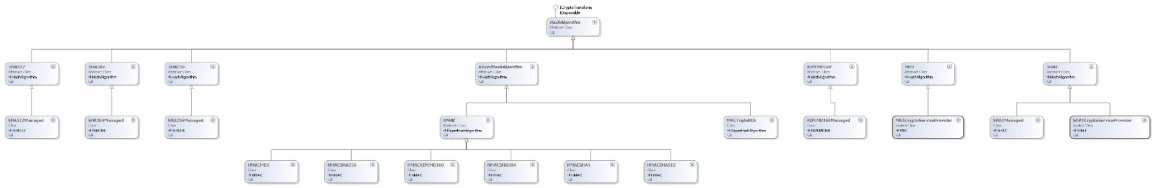

Imagem 8.3 – Hierarquia das classes de *hashing*

Todos os algoritmos que implementam a técnica de *hashing* herdam, direta ou indiretamente, de uma classe abstrata chamada **HashAlgorithm**. Como podemos notar na imagem 8.3, essa classe abstrata é herdada diretamente pelas classes **SHA1**, **MD5**, **KeyedHashAlgorithm**, **RIPEMD160**, **SHA256**, **SHA384** e **SHA512**. Cada uma dessas classes possuem características que diferem uma das outras. Essas características consistem basicamente no número de *bits* que cada uma operar e todas elas tratam-se de classes abstratas que possuem a implementação básica específica para cada algoritmo e devem obrigatoriamente devem serem implementadas pelas classes concretas. Para entendermos um pouco melhor sobre cada algorimo, vamos analisar a tabela abaixo:

Algoritmo	Descrição
MD5	O **MD5 (Message-Digest algorithm 5)** é um algoritmo de *hash* de 128 *bits* unidirecional desenvolvido pela RSA Data Security, Inc., é o sucessor do **MD4**.
SHA1	A família **SHA (Secure Hash Algorithm)** é um sistema de funções criptográficas de *hash*. O primeiro membro da familia **SHA** foi publicado em 1993 e foi chamado de **SHA**. Entretanto atualmente foi deonominado por **SHA-0** para evitar confusão com os seus sucesores. Dois anos mais tarde, o primeiro sucessor do **SHA-0** foi publicado com o nome de **SHA-1**. Existem atualmente quatro variações deste algoritmo, que se difrerenciam nas interações e na sua saída, que foram melhorados. São eles: : **SHA-224**, **SHA-256**, **SHA-384**, e **SHA-512** (Todos eles são referenciados como **SHA-2**). Já existem registros de ataques ao **SHA-0** e ao **SHA-1**, mas nenhum registro ao **SHA-2**.
HMAC	**HMAC (Hash-based Message Authentication Code)** é ma técnica que utiliza uma função de *hash* criptográfica, uma mensagem e uma chave. Constitui um dos meios predominantes para garantir que os dados não foram corrompidos durante a transição da mensagem entre o emissor e o

	receptor. Qualquer algoritmo de *hashing* pode ser utilizado com **HMAC**, sendo preferíveis as funções mais seguras, como por exemplo a **SHA-1**.

Abaixo será exibido um exemplo para cada algoritmo que vimos na tabela acima. Basicamente, todas as classes que fornecem implementações de algoritmos de *hashing*, possuem um método chamado *ComputeHash* que recebem um *array* de *bytes* contendo a mensagem a ser aplicado o *hash*, devolvendo também um *array* de *bytes*, só que contendo o resultado do *hash*.

Vamos iniciar pelo algortimo **MD5**. Assim como as classes de criptografias simétricas e assimétricas, as classes de *hashing* também possuem seus *CSPs* correspondentes e, no caso do algoritmo **MD5**, temos uma classe abstrata chamada **MD5**, que é a classe base para todos os algoritmos **MD5** e a classe **MD5CryptoServiceProvider** que é o *CSP* correspondente. Abaixo analisaremos o código responsável por aplicar um algoritmo *hash* **MD5**:

VB.NET
```
Imports System
Imports System.Security.Cryptography

Using csp As New MD5CryptoServiceProvider
    Dim msg As Byte() = Encoding.Default.GetBytes("MinhaSenha")
    Dim hash As Byte() = csp.ComputeHash(msg)

    For i As Integer = 0 To hash.Length - 1
        Console.Write(hash(i).ToString("x2"))
    Next
End Using
```

C#
```
using System;
using System.Security.Cryptography;

using (MD5CryptoServiceProvider csp = new
MD5CryptoServiceProvider())
{
    byte[] msg = Encoding.Default.GetBytes("MinhaSenha");
    byte[] hash = csp.ComputeHash(msg);

    for (int i = 0; i < hash.Length; i++)
        Console.Write(hash[i].ToString("x2"));
}
```

Como falamos acima, invocamos o método ComputeHash passando o valor onde será aplicado o *hash* e é devolvido um *array* de *bytes* contendo o valor do *hash* gerado. Apenas fazemos um laço *For* para converter cada *byte* em valor *hexadecimal* (isso é determinado pela sobrecarga do método *ToString*) correspondente para visualizarmos a valor, que é "775b70588a139c914412b7fdbc20d1c7" e este valor será sempre o mesmo para "MinhaSenha".

Utilizando agora o algoritmo **SHA1**, o código muda ligeiramente. Apenas o que irá mudar será o *CSP* utilizado. Só que para este algoritmo, temos dois *CSPs* disponíveis que são **SDA1CryptoServiceProvider** e o **SHA1Managed**. O resultado de ambos são idênticos, mas o que muda é seu comportamento interno, ou seja, a classe **SHA1Managed** é uma versão gerenciada do algoritmo **SHA1**, o que proporciona um gerenciamento melhor de memória, sem a necessidade de servir de *wrapper* para objetos não gerenciados, como é o caso do **SDA1CryptoServiceProvider**. Um exemplo da sua utilização é a seguinte:

```
VB.NET
Imports System
Imports System.Security.Cryptography

Using alg As New SHA1Managed
    Dim msg As Byte() = Encoding.Default.GetBytes("MinhaSenha")
    Dim hash As Byte() = alg.ComputeHash(msg)

    For i As Integer = 0 To hash.Length - 1
        Console.Write(hash(i).ToString("x2"))
    Next
End Using

C#
using System;
using System.Security.Cryptography;

using (SHA1Managed alg = new SHA1Managed())
{
    byte[] msg = Encoding.Default.GetBytes("MinhaSenha");
    byte[] hash = alg.ComputeHash(msg);

    for (int i = 0; i < hash.Length; i++)
        Console.Write(hash[i].ToString("x2"));
}
```

O valor do *hash* gerado para a *string* "MinhaSenha" através da classe **SHA1Managed** é a seguinte: "16c5e5376a4654937efedc675e941e32f917985e". Um detalhe importante aqui é que, se utilizarmos o *CSP* **SHA1CryptoServiceProvider** o resultado seria o mesmo.

Finalmente, iremos utilizar o algoritmo **HMAC**. Através da imagem 8.3, a classe abstrata **HMAC** é a classe base para todas as implementações deste algoritmo e estende a classe **KeyedHashAlgorithm**. Contrariamente às assinaturas digitais, os MAC's são computados e verificados com a mesma chave, para que assim, possam ser apenas verificadas pelo respectivo receptor. Um cenário de exemplo é: A Empresa 1 (emissora da mensagem) e a Empresa 2 (receptora) compartilham uma mesma chave secreta. A Empresa 1 utiliza a mensagem e a chave para computar o MAC, e envia o MAC juntamente com a mensagem. Quando a Empresa 2 receber a mensagem, descodifica o MAC e verifica-o para ver se o seu MAC corresponde ao da Empresa 1. Se corresponder, então ele sabe que a mensagem é da Empresa 1 e que ninguém a modificou desde que foi enviada pela Empresa 1. Se alguém tentar corromper a mensagem, pode fazê-lo, mesmo não possuindo a chave secreta, contudo não consegue produzir o MAC. Neste caso, o receptor irá detectar a alteração que foi feita e sabe que a mensagem foi corrompida.

Os algoritmos **HMAC** utilizam uma chave gerada randomicamente para aplicar o *hash*. Essa chave é bem semelhante ao *Salt* e é concatenada com o valor de *hash* gerado da mensagem a ser enviada. O resultado da concatenação entre a chave e o *hash* da mensagem é novamente submetido a ao algoritmo de *hash*, resultando em um valor duas vezes "hasheado" com um *Salt*. Como a chave é gerada randomicamente, a cada geração, o *output* para uma mesma mensamgem é completamente diferente.

Nota: *Salt* são utilizados para dificultar o processo de *hashing*, ou seja, são valores adicionais, geralmente gerados randomicamente, que são concatenados na mensagem que irá sofrer o processo de *hashing*. Quando você manipula dados que estão sendo armazenados dentro de um banco de dados qualquer em seu formato "*hashed*", você precisa também armazenar o *Salt* gerada, justamente para conseguir comparar o *hash* informado pelo usuário com o *hash* armazenado dentro da base de dados.

Existem várias implementações dentro do .NET Framework para **HMAC**, cada uma utilizando um algoritmo diferente, geralmente diferenciando a quantidade de *bits* em seu valor *hash* criado. Para fins de exemplo, vamos utilizar a classe **HMACSHA256** e, em relação ao código que vimos um pouco mais acima, a única alteração é apenas o nome do algoritmo:

```
VB.NET
Imports System
Imports System.Security.Cryptography

Using alg As New HMACSHA256
    Dim msg As Byte() = Encoding.Default.GetBytes("MinhaSenha")
    Dim hash As Byte() = alg.ComputeHash(msg)

    For i As Integer = 0 To hash.Length - 1
        Console.Write(hash(i).ToString("x2"))
    Next
```

```
End Using

C#
using System;
using System.Security.Cryptography;

using (HMACSHA256 alg = new HMACSHA256())
{
    byte[] msg = Encoding.Default.GetBytes("MinhaSenha");
    byte[] hash = alg.ComputeHash(msg);

    for (int i = 0; i < hash.Length; i++)
        Console.Write(hash[i].ToString("x2"));
}
```

Vale lembrar que o valor do *hash* criado para o mesmo valor será sempre alterado, devido a natureza do algoritmo, como já vimos acima. Um detalhe importante é que as classes que fornecem algoritmos de **HMAC** possuem uma propriedade chamada *Key*, que retorna um *array* de *bytes*, representando a chave utilizada no algoritmo *hash*.

DPAPI

Com o lançamento do Windows 2000, uma nova API de proteção de dados foi criada. Essa API é chamada de **DPAPI (Data Protection API)** e encapsula grande parte das complexidades que estão em torno dos processos de criptografia e descriptografia.

Essas classes estão integradas com o Windows, mas não são gerenciadas pelo .NET Framework. Na versão 2.0 do .NET Framework, foram introduzidas duas novas classes chamadas **ProtectedData** e **ProtectedMemory** que servem como *wrapper* para as classes contidas dentro da **DPAPI** e tornam a sua utilização bastante simples. Essas classes estão contidas dentro do *namespace* **System.Security.Cryptography** mas exige uma referência adicional para a *System.Security.dll*, fornecida com a versão 2.0 do .NET Framework.

A primeira delas, **ProtectedData** é utilizada para proteger dados definidos pelo usuário. Essa classe não exige outras classes (algoritmos) de criptografia. Essa classe fornece dois métodos principais chamados de *Protect* e *Unprotect*. Ambos os métodos são estáticos e, o método *Protect*, além de outros parâmetros, recebe principalmente um *array* de *bytes* que representa o valor a ser protegido, retornando também um *array* de *bytes* contendo o valor criptografado; já o método *Unprotect* recebe um *array* de *bytes* representando o dado protegido (criptografado) e retorna um *array* de *bytes* contendo o conteúdo em seu formato original.

É possível utilizar uma entropia quando invocamos o método *Protect*. Essa entropia trata-se de um valor aleatório, fornecido pela aplicação, que será utilizada pelo **DPAPI** na formação da chave de criptografia. O problema com o uso de um parâmetro adicional de

entropia é precisar ser armazenado com segurança pelo aplicativo, o que apresenta outro problema de gerenciamento de chaves, como já discutimos acima. É importante dizer também que, se um valor de entropia é passado para o método *Protect*, o mesmo valor deve ser também passado para o método *Unprotect*.

Finalmente, ambos os métodos recebem em seu último parâmetro um enumerador do tipo **DataProtectionScope**. Esse enumerador fornece duas opções, a saber:

Opção	Descrição
CurrentUser	O data a ser protegido é associado com o usuário corrente e, somente as *threads* que estão rodando com esse usuário é que poderão descriptografar os dados.
LocalMachine	O data a ser protegido é associado com a máquina. Qualquer processo rodando dentro do computador, poderá descriptografar os dados.

O código abaixo exemplifica a utilização da classe **ProtectedData**, onde devemos criar uma entropia (pode ser opcional *null* em Visual C# e *Nothing* em Visual Basic .NET) e submetermos tanto a entropia quanto a mensagem a ser criptografada para os métodos *Protect* e *Unprotect*:

VB.NET
```
Imports System
Imports System.Security.Cryptography

Dim entropia() As Byte = {12, 73, 6, 92}
Dim msg As String = "Valor a ser protegido."
Dim msgEmBytes() As Byte = Encoding.Default.GetBytes(msg)
Dim msgProtegida() As Byte = _
    ProtectedData.Protect( _
        msgEmBytes, _
        entropia, _
        DataProtectionScope.CurrentUser)

Console.WriteLine( _
    Encoding.Default.GetString( _
        ProtectedData.Unprotect( _
            msgProtegida, _
            entropia, _
            DataProtectionScope.CurrentUser)))
```

C#
```
using System;
using System.Security.Cryptography;

byte[] entropia = { 12, 73, 6, 92 };
string msg = "Valor a ser protegido.";
byte[] msgEmBytes = Encoding.Default.GetBytes(msg);
```

```
byte[] msgProtegida =
    ProtectedData.Protect(
        msgEmBytes,
        null,
        DataProtectionScope.CurrentUser);

Console.WriteLine(
    Encoding.Default.GetString(
        ProtectedData.Unprotect(
            msgProtegida,
            null,
            DataProtectionScope.CurrentUser)));
```

Ainda dentro do **DPAPI** temos a classe **ProtectedMemory**. Como o próprio nome diz, essa classe protege os dados que residem na memória e, assim como a classe **ProtectedData**, não necessita de outras classes (algoritmos) de criptografia.

As diferenças entre a classe **ProtectedData** e a classe **ProtectedMemory** são:

- A classe **ProtectedMemory** não utiliza entropia;
- O enumerador que é passado para os métodos *Protect* e *Unprotect* da classe **ProtectedMemory** é do tipo **MemoryProtectionScope**, que contém as seguintes opções:

Opção	Descrição
CrossProcess	Todo código em qualquer processo pode descriptografar os dados.
SameLogon	Somente o código que está rodando dentro do mesmo contexto de usuário que protegeu os dados poderá descriptografar os dados.
SameProcess	Somente o código que está rodando dentro do mesmo processo que protegeu os dados poderá descriptografar os dados.

- Os métodos *Protect* e *Unprotect* não retornam nenhum valor. Ele utilizará o mesmo local onde está o *array* de *bytes* com o valor a ser protegido e aplicará a criptografia no mesmo lugar;
- Os dados que serão protegidos devem ter 16 *bytes* ou ser múltiplo de 16 *bytes*.

O código abaixo exibe a utilização da classe **ProtectedMemory**:

```
VB.NET
Imports System
Imports System.Security.Cryptography

Dim msg As String = "1234567890poiuyt"
Dim msgEmBytes() As Byte = Encoding.Default.GetBytes(msg)
ProtectedMemory.Protect(msgEmBytes,
```

```
MemoryProtectionScope.SameLogon)
ProtectedMemory.Unprotect(msgEmBytes,
MemoryProtectionScope.SameLogon)

C#
using System;
using System.Security.Cryptography;

string msg = "1234567890poiuyt";
byte[] msgEmBytes = Encoding.Default.GetBytes(msg);
ProtectedMemory.Protect(msgEmBytes,
MemoryProtectionScope.SameLogon);
ProtectedMemory.Unprotect(msgEmBytes,
MemoryProtectionScope.SameLogon);
```

Valores Randômicos

Quando utilizamos os processos de criptografia e *hashing*, em alguns momentos é necessário criarmos chaves ou valores randômicos que são necessários para garantir que o algoritmo funcione como desejado. Muitas vezes esses valores são gerados automaticamente pelos algoritmos de criptografia/*hashing*, mas o .NET Framework disponibiliza publicamente duas classes que podemos utilizar para a geração destes valores randômicos.

As classes fornecidas são: **RandomNumberGenerator** e **RNGCryptoServiceProvider**. A primeira delas, é uma classe abstrata que é a base necessária que todas as classes geradoras de valores randômicos devem herdar. Já a segunda classe, trata-se de uma classe concreta (*CSP*), chamada **RNGCryptoServiceProvider**. Ela extende a classe abstrata **RandomNumberGenerator** e implementa um **Random Number Generator (RNG)**. A imagem abaixo exibe a hierarquia entre essas classes:

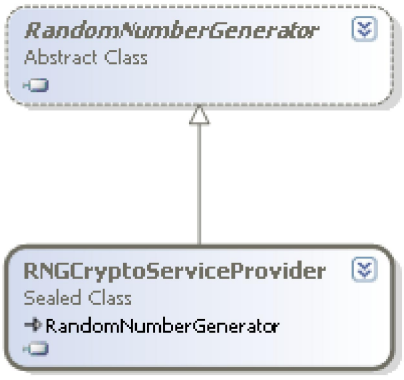

Imagem 8.3 – Hierarquia das classes que geram valores randômicos

A classe **RNGCryptoServiceProvider** é extensamente utilizada pelo próprio .NET Framework na geração de chaves e dos vetores que discutimos acima, durante a geração

automática desses valores. Essa classe fornece dois métodos chamados *GetBytes* e *GetNonZeroBytes*. O método *GetBytes* recebe um *array* de *bytes* onde ele irá populá-lo com uma seqüencia randômica de valores. Assim, como o método *GetBytes*, o método *GetNonZeroBytes*, também recebe um *array* de *bytes* onde o método irá populá-lo com uma seqüencia randômica, só que sem zeros. O código abaixo exibe uma possível forma de como utilizar essa classe:

```
VB.NET
Imports System
Imports System.Security.Cryptography

Dim csp As New RNGCryptoServiceProvider
Dim salt(64) As Byte
csp.GetBytes(salt)

C#
using System;
using System.Security.Cryptography;

RNGCryptoServiceProvider csp = new RNGCryptoServiceProvider();
byte[] salt = new byte[64];
csp.GetBytes(salt);
```