



# La sécurité du .NET Framework

## Sommaire

La sécurité du .NET Framework .....	1
1    Introduction.....	3
2    Gestion de la sécurité de la CLR .....	4
2.1    Préambule .....	4
2.2    Approfondissement de la sécurité coté application .....	6
2.3    Paramétrage de la sécurité de la CLR.....	10
2.3.1    Les jeux d'autorisations .....	12
2.3.2    Les groupes de codes .....	16
2.3.3    Applications approuvées .....	23
2.3.4    Le CAS et les permissions d'hôte .....	23
2.4    Configuration de sécurité en ligne de commande .....	23
3    Gestion de la sécurité du programme.....	24
3.1    Restrictions d'exécution au niveau de l'assembly.....	24
3.2    Modification des droits au niveau d'une classe/méthode.....	30
3.3    Effectuer des demandes groupées.....	37
4    Sécurisation des accès.....	40
4.1    Authentification.....	41
4.1.1    Authentification avec WindowsIdentity.....	41
4.1.2    Création d'une classe d'identité personnalisée.....	42
4.1.3    Authentification d'un groupe d'utilisateurs .....	42
4.1.4    Création de classe d'identité de groupe personnalisée .....	45
4.1.5    Authentification générique .....	45
4.2    Autorisation .....	47
5    Gestion de la RBS Windows.....	49
5.1    Rappel sur le système RBS Windows.....	49
5.2    Gestion des droits via le code .....	51
6    La cryptographie dans le .NET .....	54
6.1    Empreintes des données .....	54
6.2    Chiffrement symétrique .....	57
6.2.1    Outil annexe : La classe Rfc2898DeriveBytes.....	57
6.2.2    Le chiffrement symétrique en pratique .....	59

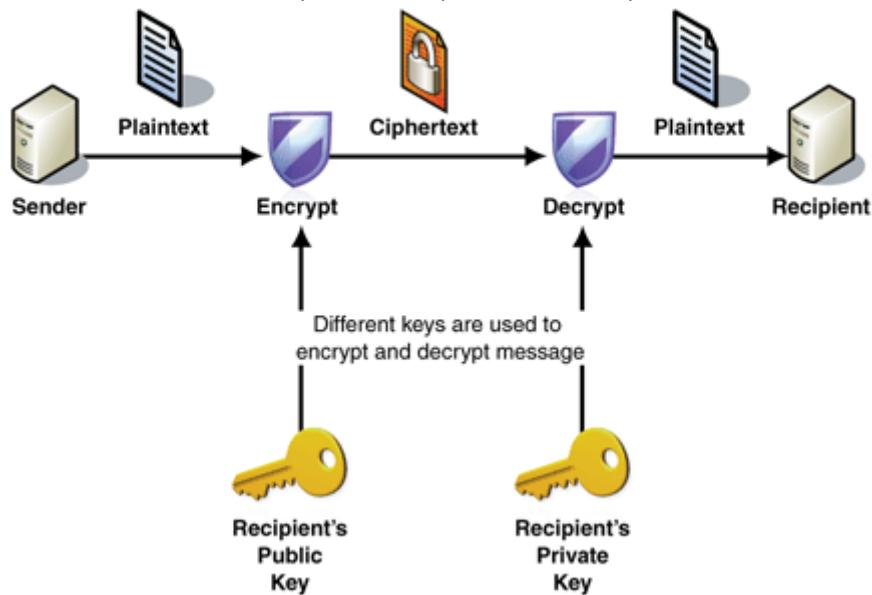
6.3	Chiffrement asymétrique .....	63
6.3.1	RSACryptoServiceProvider .....	64
6.3.2	DSACryptoServiceProvider .....	69
6.4	Signatures numériques.....	70
7	Conclusion .....	74

Dotnet-France Association



## 1 Introduction

De nos jours, la sécurité est un des piliers clé de toutes applications en entreprise. En effet, on ne pourrait pas imaginer un monde dans lequel toutes les transactions bancaires électroniques ne soient pas effectuées en utilisant des systèmes de protections adaptés.



Même si avec les solutions actuelles, la sécurité informatique absolue n'existe pas, il existe de nombreux moyens permettant de donner du fil à retordre aux hackers qui chercheraient à exploiter vos applications industrielles.

Dans ce chapitre, nous aborderons tout d'abord la sécurité d'accès aux ressources locales (propre à la CLR). Ensuite, nous verrons comment gérer les règles de sécurité d'accès aux ressources puis nous terminerons le chapitre par les techniques de cryptographie implémentées par le .NET Framework.

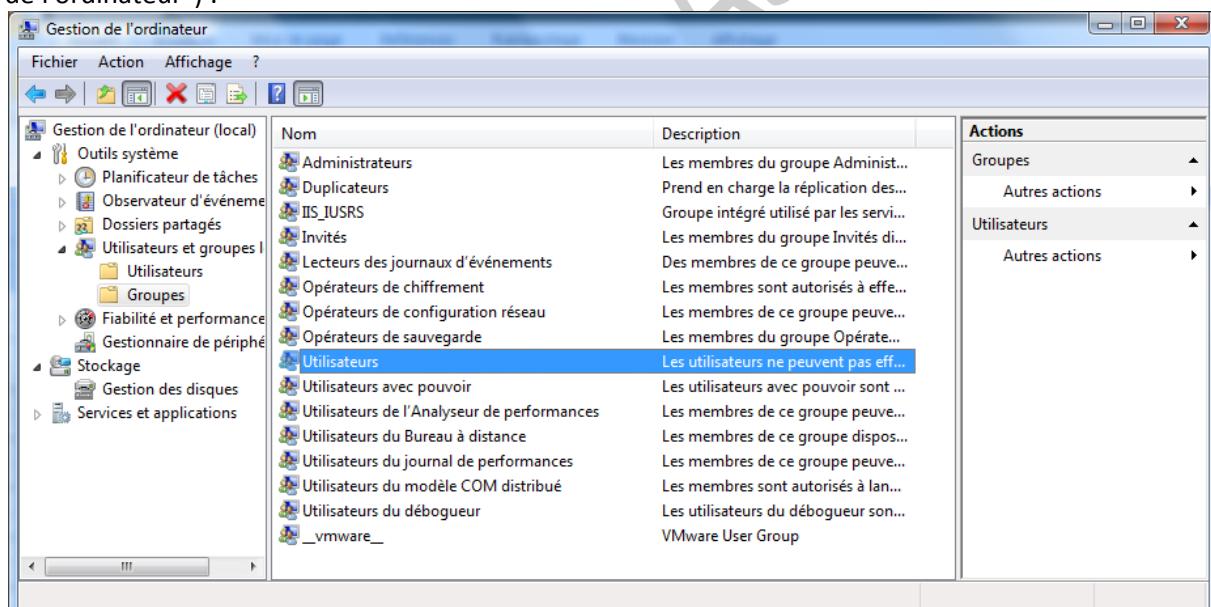
## 2 Gestion de la sécurité de la CLR

### 2.1 Préambule

Sur tous les systèmes Windows, les personnes qui utilisent les machines sont représentées par une entité virtuelle appelée "utilisateur" (Et l'un des moyens permettant de lier les personnes aux entités est l'utilisation d'un identifiant et d'un mot de passe). Ces utilisateurs peuvent soit avoir des droits qui leur sont propres (comme par exemple, le droit d'exécuter d'autre application, d'aller sur internet, etc.) et/ou être membre d'un "groupe d'utilisateur".

Ces groupes d'utilisateurs sont des entités qui attribuent des droits à un ensemble d'utilisateurs (Par exemple, l'utilisateur "Gérant" et l'utilisateur "Développeur" peuvent tous les deux exécuter des applications. Plutôt que d'attribuer ce droit à chaque utilisateur, nous utiliserons un groupe d'utilisateur pouvant exécuter des applications et nous feront en sorte que les deux utilisateurs soient liés au groupe qui nous intéresse).

Par défaut, Windows est installé avec une certaine liste de groupe dont chacun possède un rôle bien défini. Par exemple, sur une machine équipée de Windows Vista Business, nous avons les groupes suivants (Vous pouvez accéder à la gestion simple des utilisateurs et groupes en passant par "Panneau de configuration → Outils d'administration → Gestion de l'ordinateur". Vous pouvez également modifier le rôle de chacun en choisissant "Stratégie de Sécurité locale" au lieu de "Gestion de l'ordinateur") :



Ce principe de groupes/utilisateurs ayant des rôles particulier est appelé "Role-based Security" ou RBS.

Le Framework .NET possède un système comparable afin de gérer la sécurité des applications qui sont exécutées via la CLR. Ce système, en plus des possibilités via le code, est regroupé sous le nom de "Code Access Security" ou CAS en abrégé. Les assemblies pourraient être apparentés aux utilisateurs.

**Note :** Avant d'aller plus loin, il est impératif d'installer le .NET Framework 2.0 SDK disponible [ici](#). Ce SDK contient plusieurs outils de gestion du Framework .NET dont un composant enfichable de la console MMC appelé "Configuration du .NET Framework 2.0". Pour lancer l'utilitaire, cliquez le

menu Windows, allez sur "Exécuter" (ou directement dans la barre de recherche sous Vista) puis tapez "mmc". Là, faites "Fichier→Ajouter/Supprimer un composant logiciel enfichable..." et sélectionnez le composant ".NET Framework 2.0 Configuration".

**Important :** S'il est possible de gérer totalement les possibilités des applications en code managé, il est impossible d'effectuer les mêmes opérations sur du code non-managé (Cette notion de code managé ou non sera vue dans les chapitres suivants). Vous devrez vous baser sur les rôles attribués à chacun de vos utilisateurs/groupes pour éviter les problèmes. Aussi, même si cela est contraignant au début, n'oubliez jamais qu'une sécurité digne de ce nom est celle qui accorde le moins de priviléges possibles aux utilisateurs. Ce principe est bien entendu vrai en ce qui concerne les applications.

## 2.2 Approfondissement de la sécurité coté application

Si une personne peut saisir un identifiant et un mot de passe pour indiquer à la machine quel utilisateur lui attribuer, un programme ne peut pas se servir du même procédé.

Plutôt que d'utiliser un identifiant et un code de sécurité, les assemblies utiliseront des "evidence" pour s'authentifier auprès de la CLR et c'est grâce à cette evidence que la CLR va savoir dans quel groupe placer l'assembly qui demande à être exécutée.

Cette evidence peut contenir plusieurs propriétés :

- Application directory qui est le répertoire dans lequel se trouve l'assembly.
- Hash qui représente le code de hachage de l'assembly (en MD5 ou SHA1)
- Publisher qui contient la signature numérique de l'éditeur (n'est disponible que si l'assembly a été signée numériquement)
- Site, qui contient le site d'où a été téléchargé l'assembly.
- Strong name, qui représente un nom cryptographique (l'assembly doit être signée également pour avoir accès à cette propriété)
- URL qui est identique à Site. La différence est que l'URL contient le nom de l'assembly en plus du site.
- Zone qui est la zone dans laquelle l'assembly s'exécute.

Il existe deux types d'evidence :

- Les evidence d'hôte (Host evidence) qui contiennent des paramètres non modifiable.
- Les evidence d'assembly (Assembly evidence) qui contiennent des paramètres personnalisés

Coté code, vous pouvez accéder à ces informations en utilisant la propriété `AppDomain.CurrentDomain.Evidence`. Dans l'exemple suivant, nous avons créé un nouveau projet Winform dans lequel nous avons glissé une textbox configurée en "Multiline = true", "Scrollbars = Both" et "ReadOnly = true" (Optionnellement vous pouvez ajouter un écouteur sur la fenêtre principale à l'évènement `ResizeEnd`) :

```
' VB
Public Sub New()
    InitializeComponent()

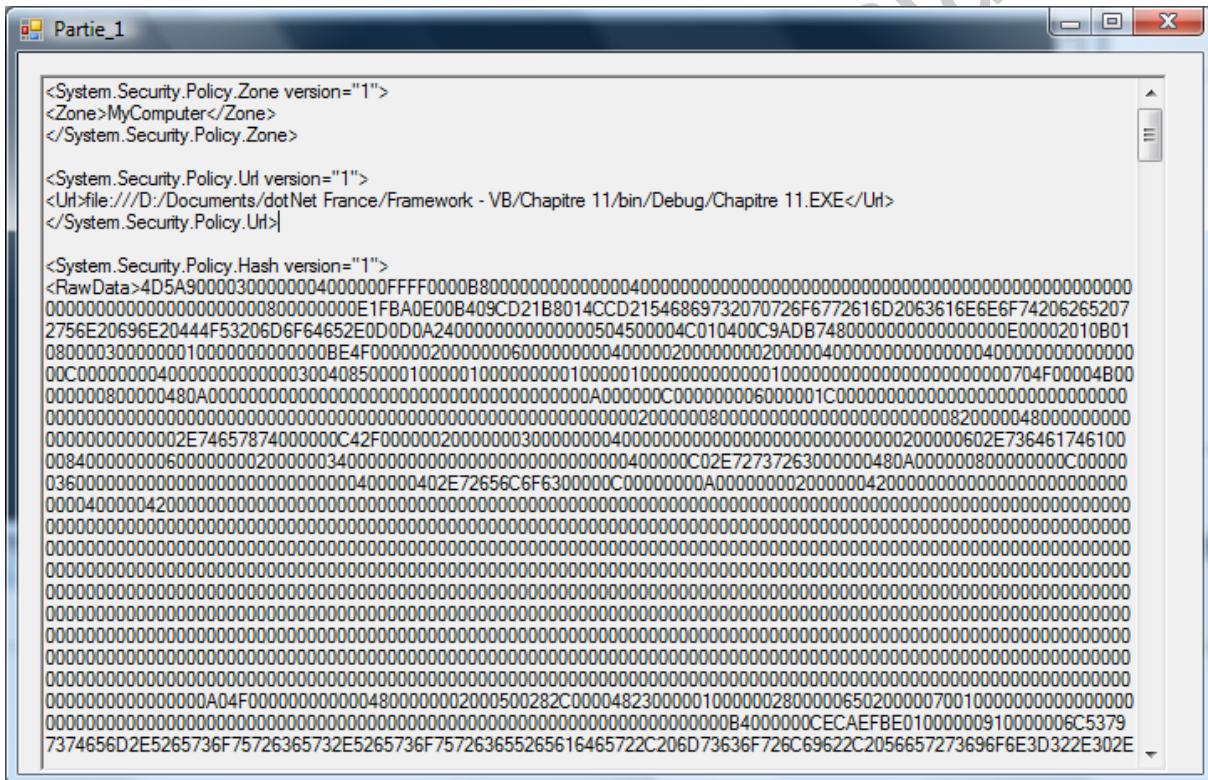
    Dim e As Evidence = AppDomain.CurrentDomain.Evidence
    For Each o As Object In e
        TextBox1.Text += o.ToString() + vbCrLf
    Next
End Sub

Private Sub Partie_1_ResizeEnd(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.ResizeEnd
    TextBox1.Size = New System.Drawing.Size(MyBase.Width - 50,
    MyBase.Height - 50)
End Sub
```

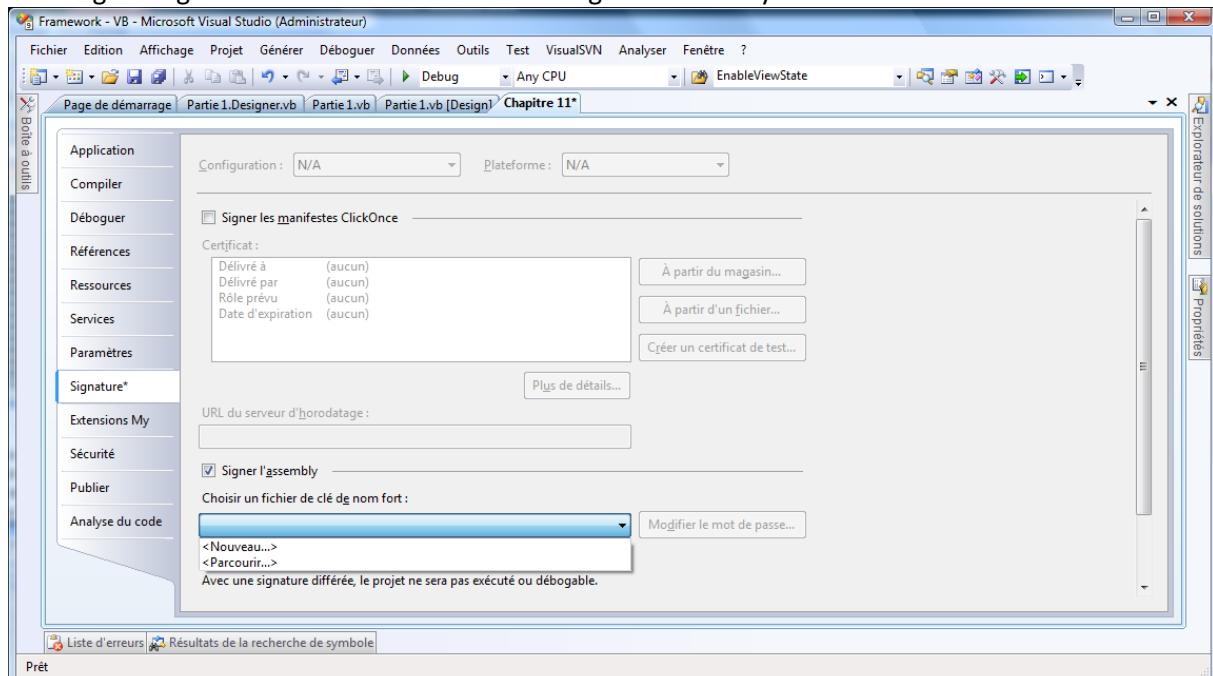
```
//C#
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        Evidence e = AppDomain.CurrentDomain.Evidence;
        foreach(Object o in e)
            textBox1.Text += o.ToString() + "\n";
    }

    private void Form1_ResizeEnd(object sender, EventArgs e)
    {
        textBox1.Size = new System.Drawing.Size(base.Width - 50,
base.Height - 50);
    }
}
```

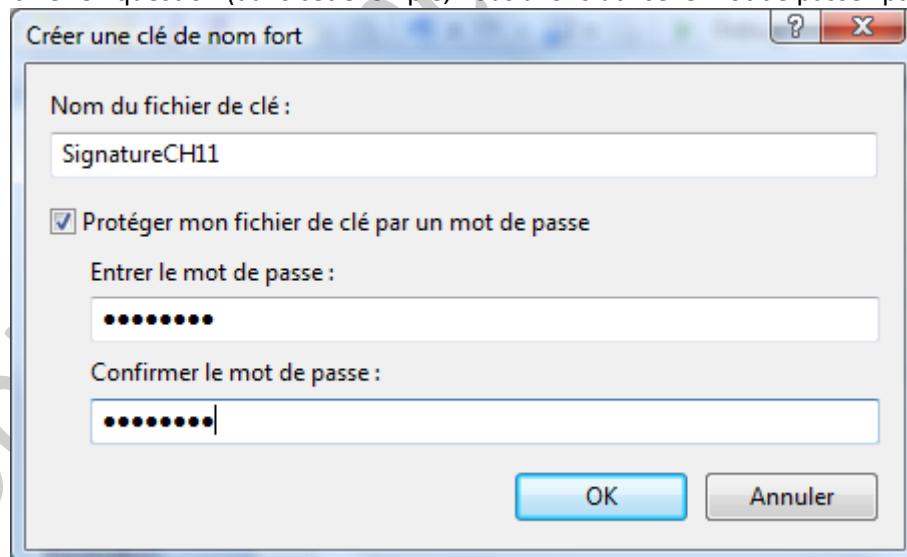
Après compilation, vous devriez une fenêtre contenant le Hash de votre application, l'URL et sa zone :



Vous avez remarqué que certaines options nécessitent la signature de l'assembly. Pour signer une assembly, cela reste relativement simple. Vous allez dans les propriétés du projet et vous allez sur l'onglet "Signature" et vous cochez la case "Signer l'assembly"



Ensuite, cliquez la liste déroulante vide et sélectionnez "<Nouveau...>". Là, vous saisissez le nom que vous souhaitez donner à votre fichier de signature ainsi qu'un éventuel mot de passe pour protéger le fichier en question (dans cet exemple, nous avons utilisé le mot de passe "password") :



Si vous retournez dans votre solution, vous avez maintenant un fichier .pfx qui est apparu et qui contient la signature numérique de votre application.

Notez que si vous ré-exécutez le projet précédent, vous avez un champ supplémentaire (StrongName) qui est apparu dans l'evidence utilisée :

Une autre information qui peut être utile pour la suite, c'est de savoir récupérer un jeton de clé publique pour votre application. Pour cela, vous devez ouvrir la commande Visual Studio et saisir la commande suivante :

sn -T "<chemin de votre assembly>"

En réponse directe, vous aurez le jeton de clé publique qui s'affichera dans la console (Pour avoir un jeton disponible, il est obligatoire de signer votre application.) :

Setting environment for using Microsoft Visual Studio 2008 x86 tools.

```
C:\Program Files\Microsoft Visual Studio 9.0\VC>sn -T "D:\Documents\dotNet Franc  
e\Framework - VB\Chapitre 11\bin\Debug\Chapitre 11.exe"
```

Microsoft (R) .NET Framework Strong Name Utility Version 3.5.30729.1  
Copyright (c) Microsoft Corporation. Tous droits réservés.

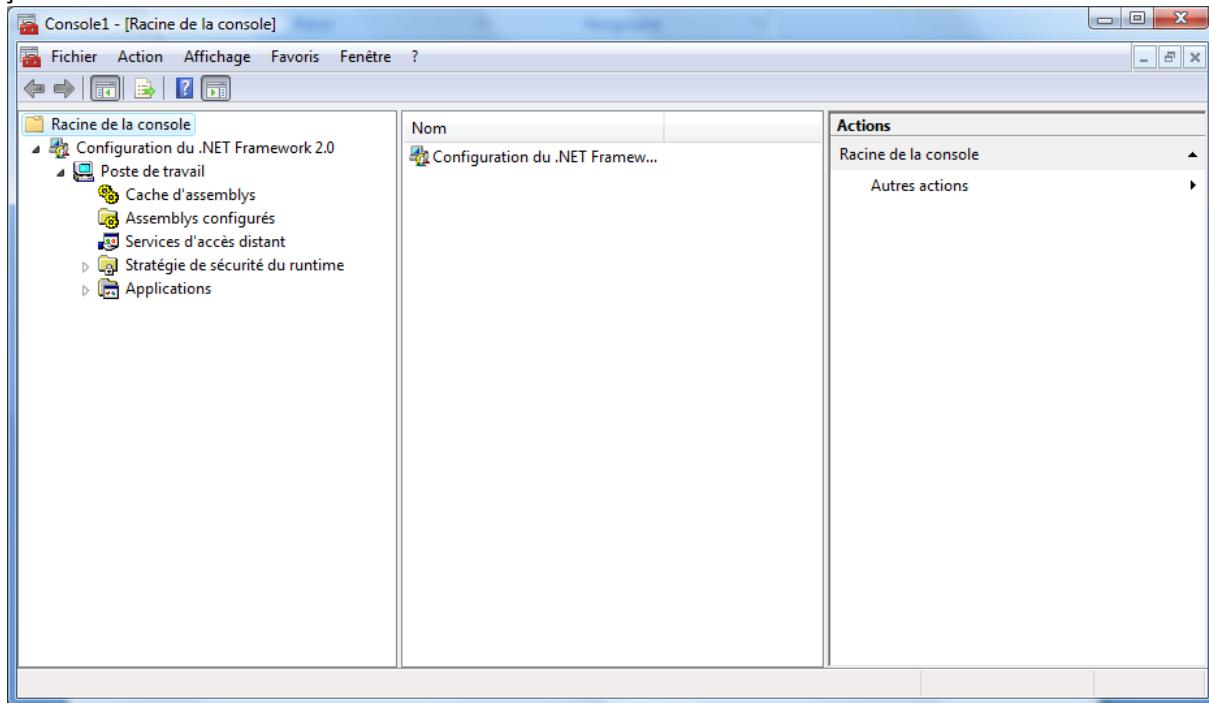
Le jeton de la clé publique est 7a507855d0c56903

C:\Program Files\Microsoft Visual Studio 9.0\VC>

## 2.3 Paramétrage de la sécurité de la CLR

Dans cette partie, nous laissons un peu de côté les règles de sécurité au niveau application pour nous intéresser plus particulièrement à la configuration de la sécurité de la CLR.

Pour commencer, nous allons ouvrir la console MMC et ajouter le composant enfichable ".NET Framework 2.0 Configuration". Ensuite, vous déployerez successivement les lignes "Configuration du Framework .NET 2.0" et "Poste de travail". Une fois cela fait, nous devrions avoir une fenêtre comme ça :



A ce stade, nous avons plusieurs lignes intéressantes :

- "Cache d'assemblys" qui permet d'ajouter/supprimer des assemblies résidentes sur la machine. Par défaut, nous y retrouvons bien sur toutes les assemblies du .NET Framework installé mais également d'autres assemblies ayant été installées par un autre logiciel (notamment, les "Microsoft.VisualStudio" ou encore les "Microsoft.Office").
- "Services d'accès distant" qui va servir à afficher la liste des paramétrages de sécurité en ce qui concerne l'accès à distance.
- "Stratégie de sécurité du runtime" qui vous permettra de gérer totalement les règles de sécurité de la CLR.
- "Applications" qui servira à lister les assemblies configurées et à créer de nouvelles règles

Dans cette première partie, nous nous intéresserons plus particulièrement à la ligne "Stratégie de sécurité du runtime".

Si vous déroulez cette ligne, vous avez trois autres champs disponibles :

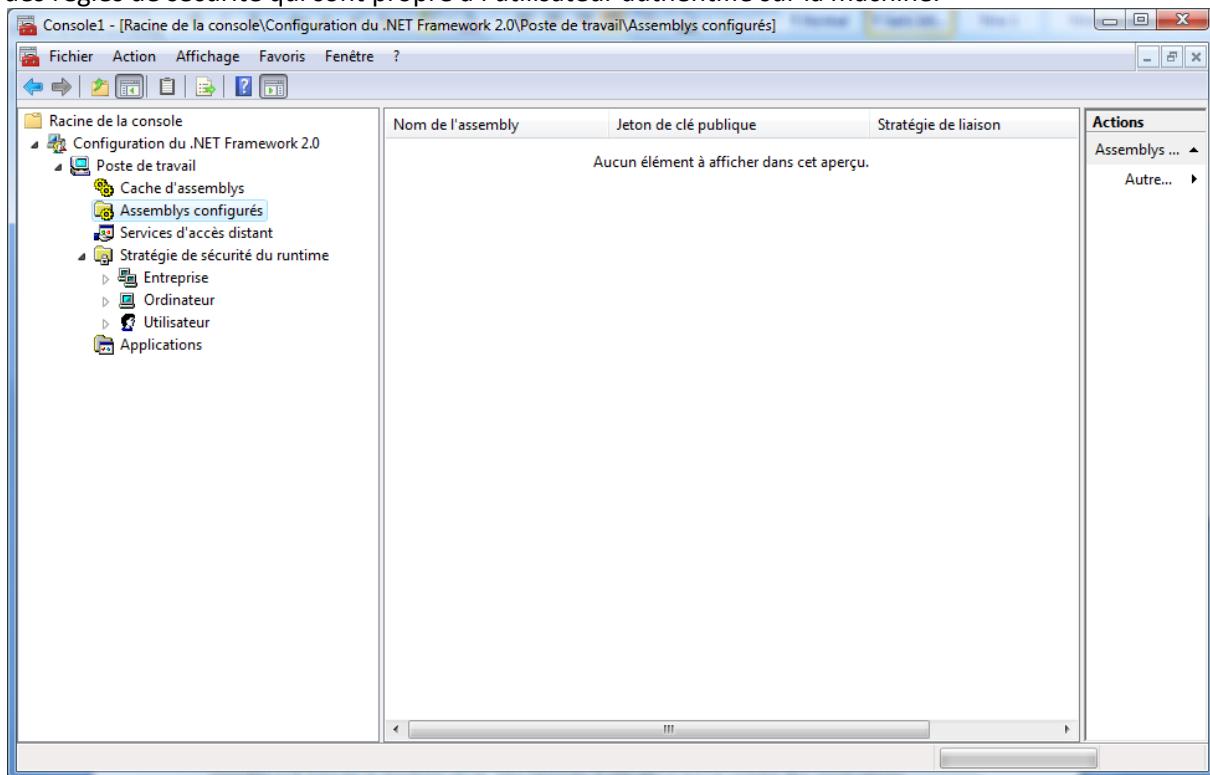
- Entreprise
- Ordinateur
- Utilisateur

Chacune de ces lignes représente en fait un niveau d'étendue des règles de sécurité qu'il contient.

La première, Entreprise, concerne toute l'infrastructure dans laquelle se trouve la machine configurée ; c'est la partie la plus globale que l'on puisse trouver.

La seconde ligne, Ordinateur, permet de définir des règles de sécurité de la CLR qui s'appliquent à toute la machine et ce, peu importe l'utilisateur qui se servira des applications.

La dernière ligne, Utilisateur, est la plus restreinte au niveau étendue. Elle permet de définir des règles de sécurité qui sont propre à l'utilisateur authentifié sur la machine.



Pour cette partie, nous allons modifier les paramètres propres à l'utilisateur en cours. En effet, c'est la seule section qui contient toutes les catégories de sécurité possible ; les autres ayant les mêmes fonctions mais à plus grande échelle.

Si vous déroulez la ligne Ordinateur, vous avez quatre nouvelles sections qui apparaissent :

- "Groupes de codes", qui permettent d'associer des jeux d'autorisations avec des assemblies particulières.
- "Jeux d'autorisations", qui sont l'équivalent des groupes d'utilisateurs. Ils rassemblent plusieurs règles de sécurité sous un même nom.
- "Assemblies de stratégie", qui contient toutes les assemblies soumises à des règles de sécurité et qui se trouvent dans le cache d'assembly (GAC pour Global Assembly Cache).

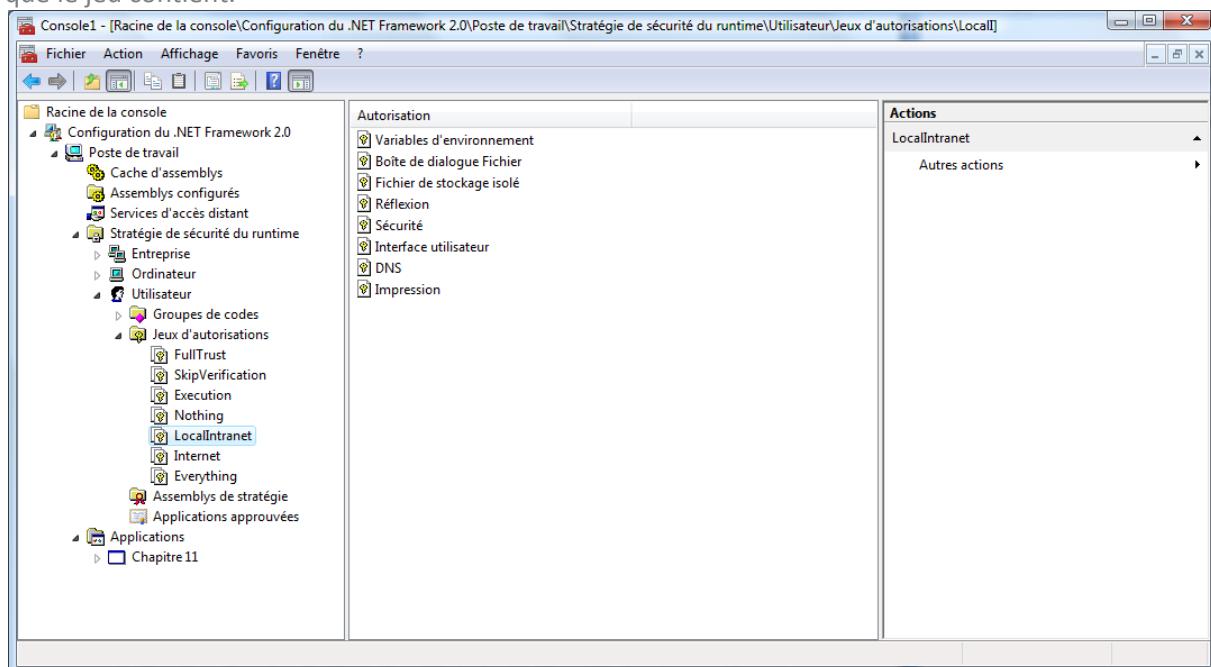
"Applications approuvées", qui contient toutes les assemblies qui sont considérées comme digne de confiance (Fully Trusted). Les autres assemblies soumises à des règles sont dites de confiance partielle (Partially Trusted).



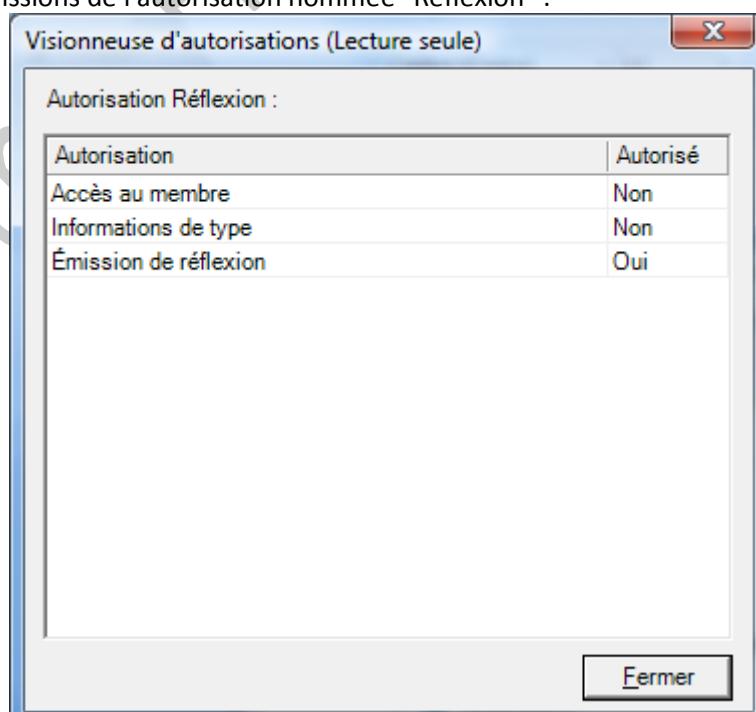
### 2.3.1 Les jeux d'autorisations

Ces jeux d'autorisations vous permettent de rassembler une ou plusieurs règle(s) de sécurité sous un même nom. Si vous déroulez cette ligne, vous pouvez remarquer que par défaut, le Framework propose sept jeux d'autorisations ; chacun d'entre eux ayant des autorisations bien spécifiques ; les autorisations contenant à leur tour un ensemble de permissions.

**Note :** Par défaut, une page d'aide est affichée quand vous cliquez sur un jeu d'autorisation. Si vous cliquez sur le lien "Afficher les autorisations", vous pouvez visualiser toutes les autorisations que le jeu contient.

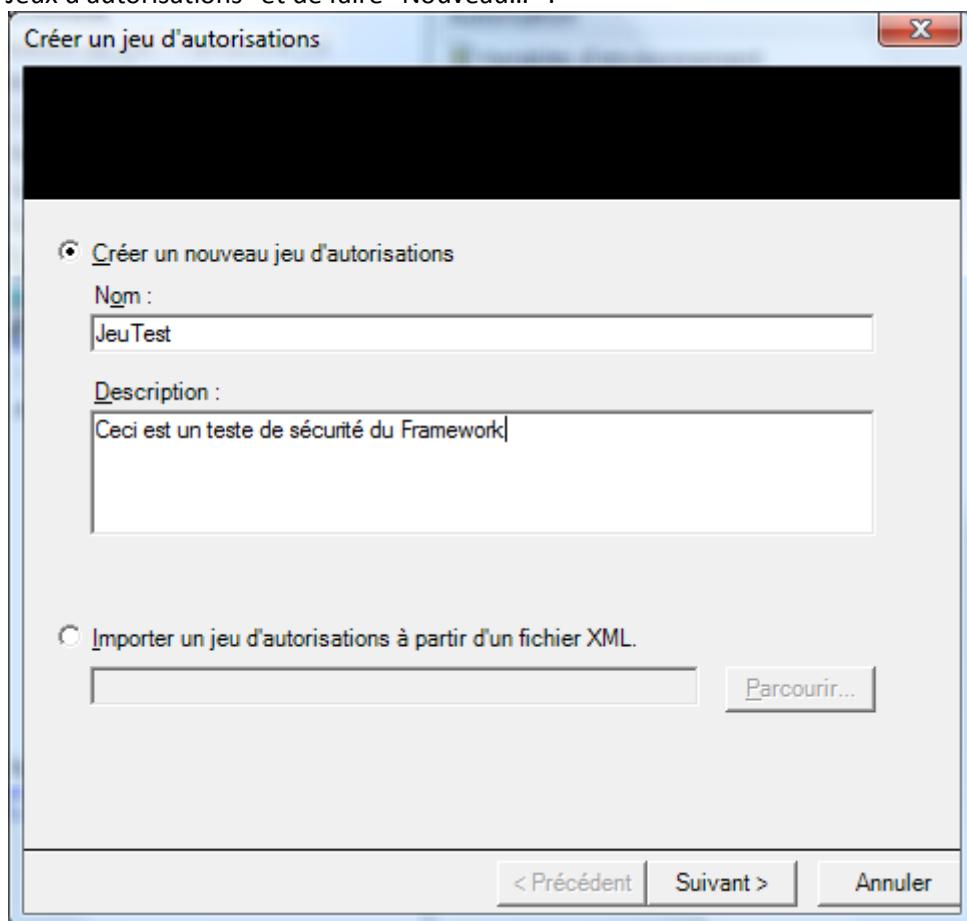


Pour voir plus en détail chaque autorisation du jeu d'autorisation, il vous suffit de double-cliquer sur l'une des autorisations affichées pour en montrer les permissions. Ci-dessous, nous affichons les permissions de l'autorisation nommée "Réflexion" :



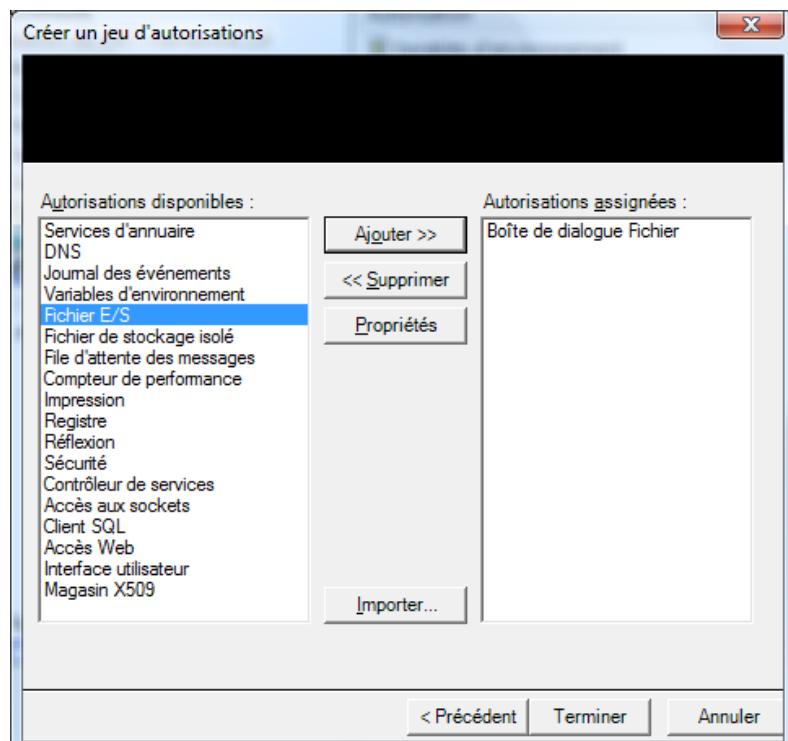


Vous pouvez bien sûr créer vos propres jeux d'autorisation. Pour ça, il vous suffit de faire clic-droit sur "Jeux d'autorisations" et de faire "Nouveau..." :



Si vous avez exporté un jeu d'autorisation en XML, vous pouvez l'importer à nouveau en sélectionnant "Importer un jeu d'autorisation à partir d'un fichier XML". Sinon, laissez la première option sélectionnée, saisissez le nom et une description du jeu d'autorisations. Ensuite, cliquez sur "Suivant".

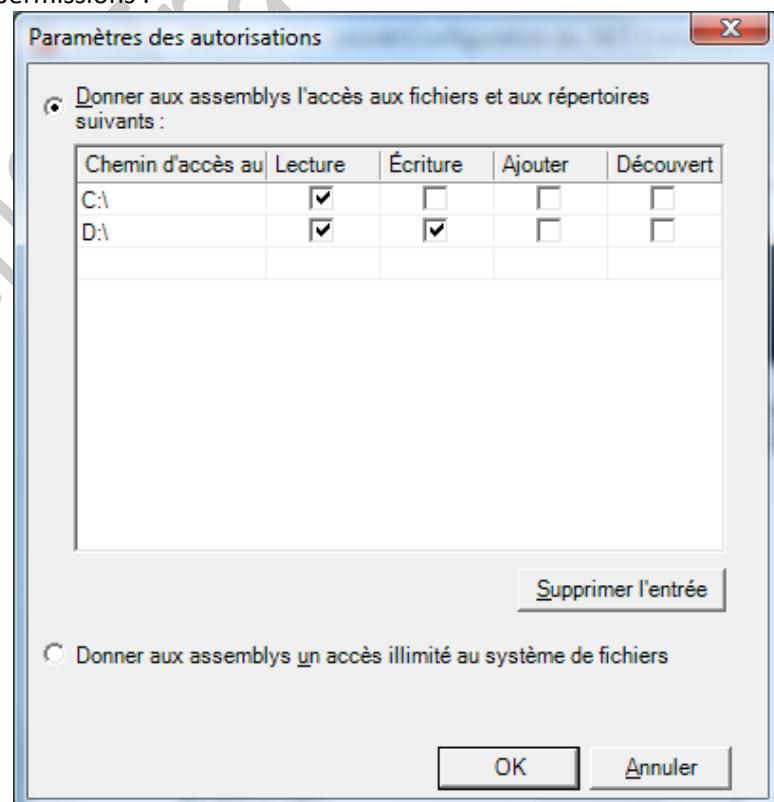
Dans le cadre suivant, vous sélectionnez les autorisations disponibles pour le jeu en cours d'édition. Pour ajouter des autorisations, il vous suffit de cliquer sur l'autorisation à ajouter et de faire "Ajouter >>".



Chaque autorisation porte un nom assez explicite qui n'a pas besoin d'être détaillé. Aussi, je me contenterais de détailler ceux qui peuvent l'être moins :

- Magasin X509 : Spécifie l'autorisation d'accéder, modifier, ajouter ou supprimer une entrée du magasin de certificats [X509](#).
- Service d'annuaire : Spécifie l'accès aux bases de données [Active Directory](#).
- DNS : Spécifie les autorisations effectives quand à l'envoi de requêtes DNS.

Lorsque vous cliquez sur le bouton ajouter, une nouvelle fenêtre s'ouvre, dans laquelle vous pouvez modifier les permissions :





Dans cet exemple, nous avons ajouté les permissions suivantes :

- Accès en lecture sur le disque C: et en lecture/écriture sur le disque D: (Fichier E/S)
- Autorisation d'afficher les boîtes de dialogue pour ouvrir un fichier. (Boîte de dialogue Fichier)
- Accès total aux interfaces utilisateurs. (Interface Utilisateur)
- Accès total à la gestion de la sécurité de la CLR. (Sécurité)

Vous pouvez bien sûr modifier les permissions déjà ajoutées en les sélectionnant dans la colonne de droite et en cliquant sur le bouton "Propriétés".

Une fois que tout est terminé, cliquez sur "Terminer" pour valider votre jeu d'autorisation. Il apparaît ensuite dans la liste des jeux d'autorisations disponibles.

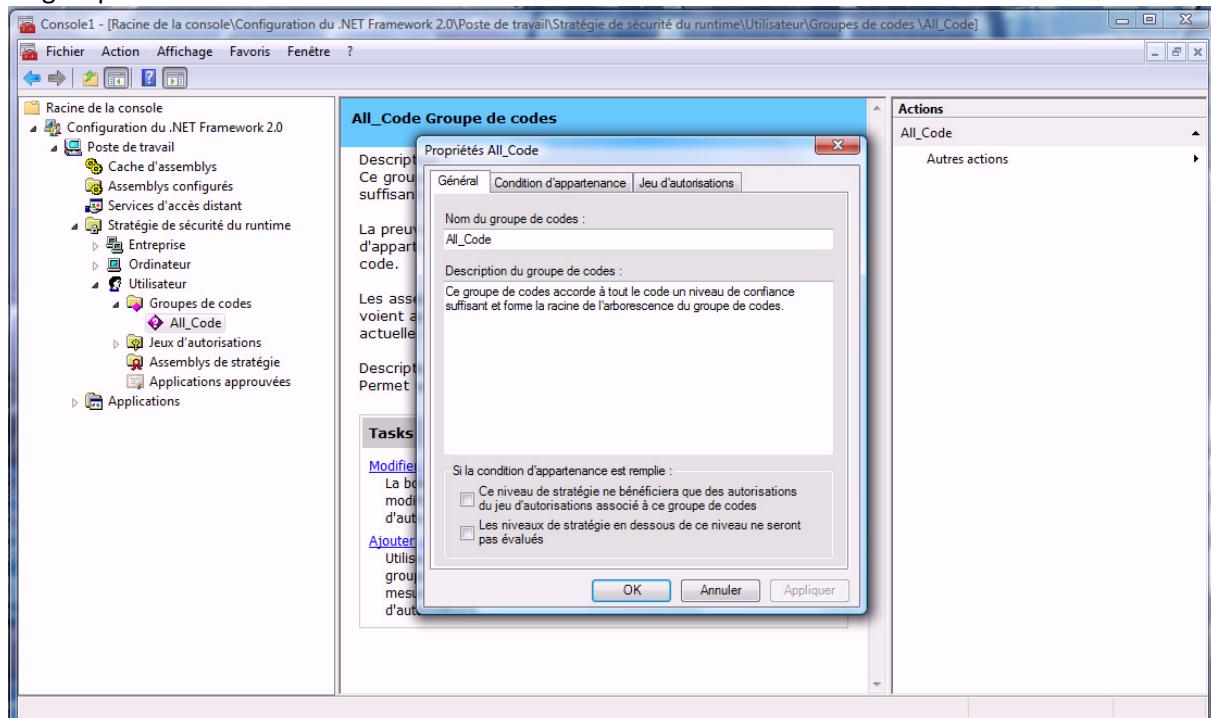


## 2.3.2 Les groupes de codes

### 2.3.2.1 Configuration des groupes de codes

Les groupes de codes permettent donc de lier des assemblies à un ou plusieurs jeux (x) d'autorisations. Si vous déroulez cette ligne, vous aurez toujours un groupe de code par défaut nommé "All\_code" que vous ne pouvez pas supprimer. Ce groupe de code concerne toutes les assemblies exécutées et leur accorde le jeu d'autorisation "FullTrust" (Qui est équivalent à placer les assemblies dans la section "Applications approuvées").

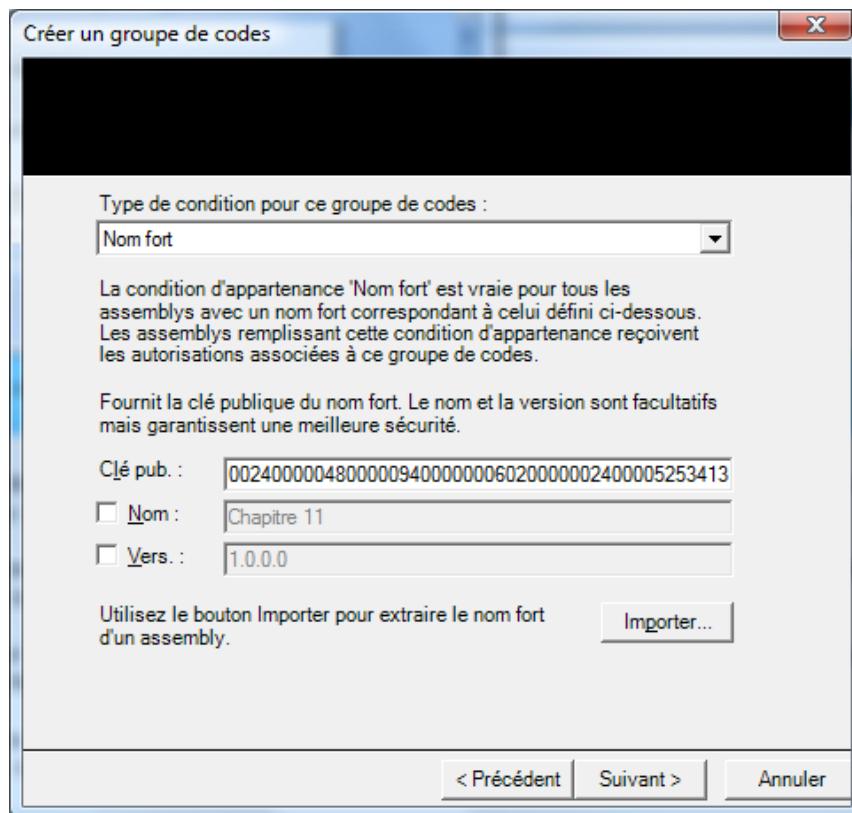
Dans le texte qui s'affiche quand All\_code est sélectionné, cliquez sur "modifier les propriétés du groupe de codes" :



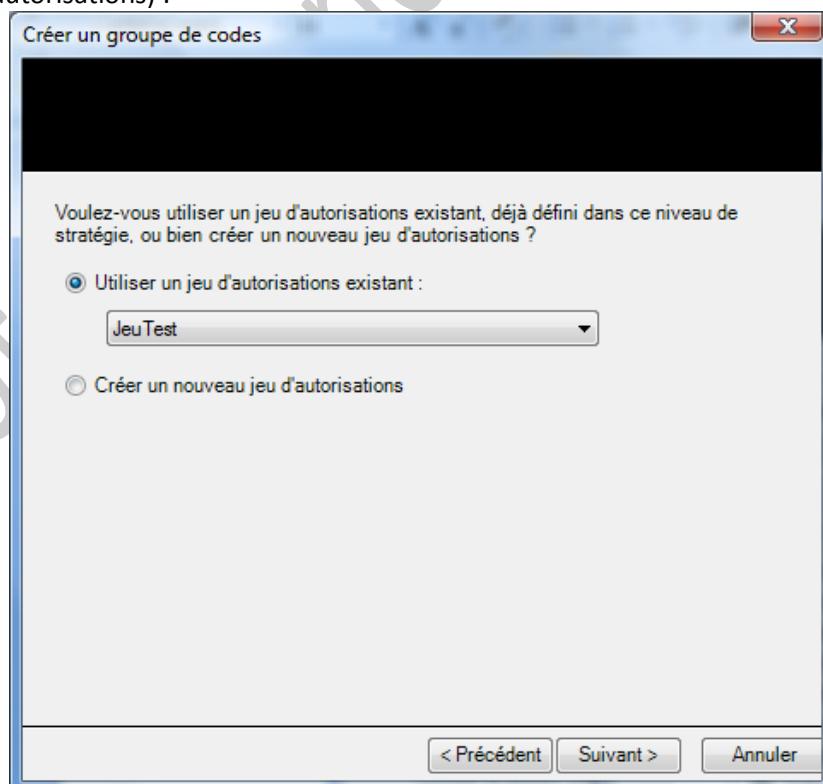
Vous avez 3 onglets à votre disposition :

- Général, permettant de modifier la description ou le nom ainsi que deux autres options du groupe de code.
- Condition d'appartenance, qui fixe quel paramètre sera utilisé pour déterminer quelles assemblies seront affectées par ce groupe de code.
- Jeu d'autorisation, qui permet de spécifier quel jeu d'autorisation sera utilisé pour ce groupe de code.

Là aussi, vous pouvez créer vos propres groupes de codes. Il vous suffit de faire un clic droit sur la ligne "All\_code" et de faire "Nouveau...". Comme précédemment, vous pouvez saisir le nom et une description ou importer un fichier XML. Une fois que vous avez cliqué sur Suivant, l'utilitaire va vous demander de sélectionner le paramètre qui sera recherché pour définir quelles assemblies seront affectées par ce groupe de codes. Ci-dessous, nous avons sélectionné le tri par nom fort du programme et nous spécifions celui de notre assembly en utilisant le bouton "Importer" :



Ensuite, cliquez sur "Suivant". Sur la fenêtre suivante, vous sélectionnerez le jeu d'autorisations qui régira les permissions du groupe de code (Vous pouvez aussi choisir de créer un nouveau jeu d'autorisations) :





Dans l'exemple, nous basons notre groupe de codes sur le jeu d'autorisations créé plus haut. Quand tout est paramétré selon vos besoins, cliquez sur "Suivant" puis sur "Terminer". De toute façon, vous pourrez modifier les autorisations du groupe plus tard.

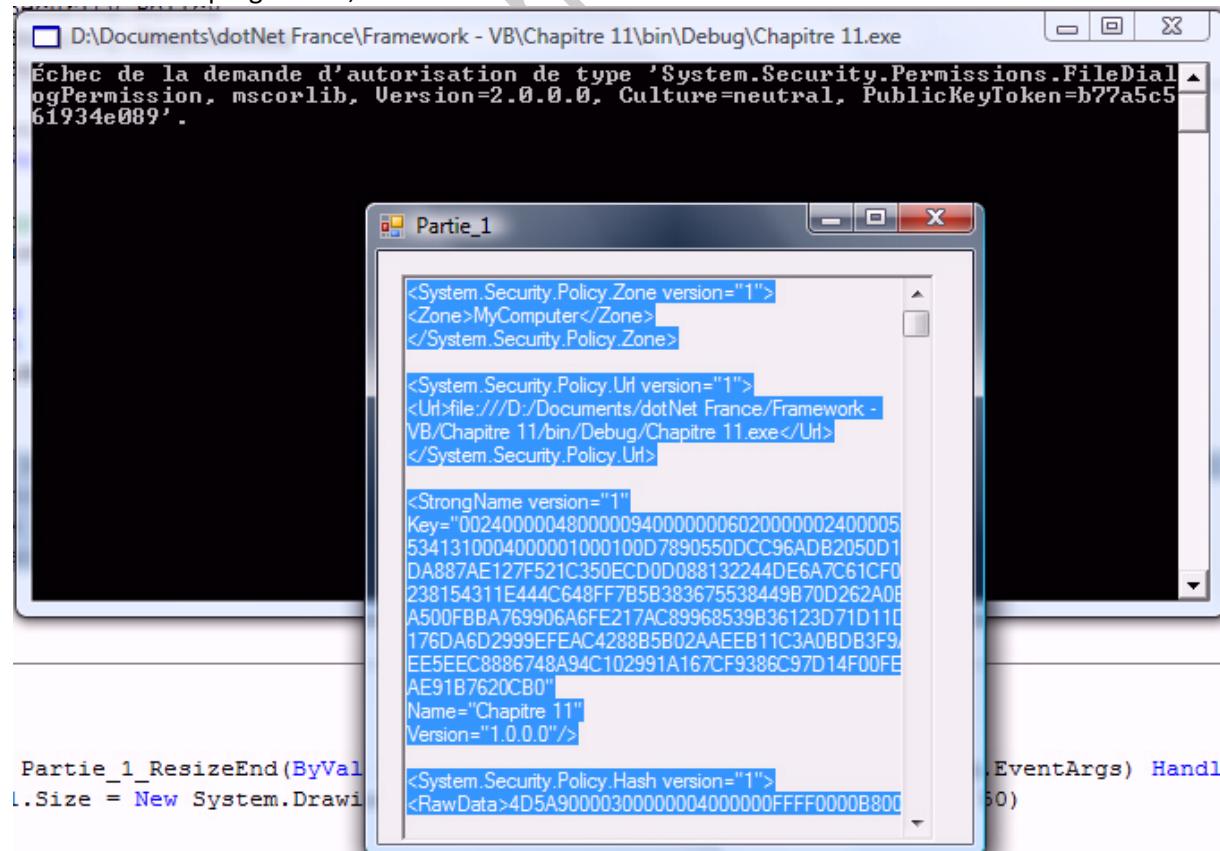
Pour l'exemple, nous modifierons le groupe de codes créé et nous cocherons la case "Ce niveau de stratégie ne bénéficiera que des autorisations du jeu d'autorisation associé à ce groupe de codes" ; la raison en sera expliquée plus bas.

Coté code, on réutilisera le projet créé au début de ce chapitre ; Nous n'enlèverons aucun code mais nous ajouterons le code suivant dans le constructeur de la classe principale, juste avant la boucle foreach :

```
'VB
Try
    Dim s As System.Windows.Forms.SaveFileDialog = New
System.Windows.Forms.SaveFileDialog()
    s.ShowDialog()
Catch ex As Exception
    Console.WriteLine(ex.Message)
End Try

//C#
try
{
    System.Windows.Forms.SaveFileDialog s = new
System.Windows.Forms.SaveFileDialog();
    s.ShowDialog();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

A l'exécution du programme, vous devriez avoir un résultat similaire à celui-ci :

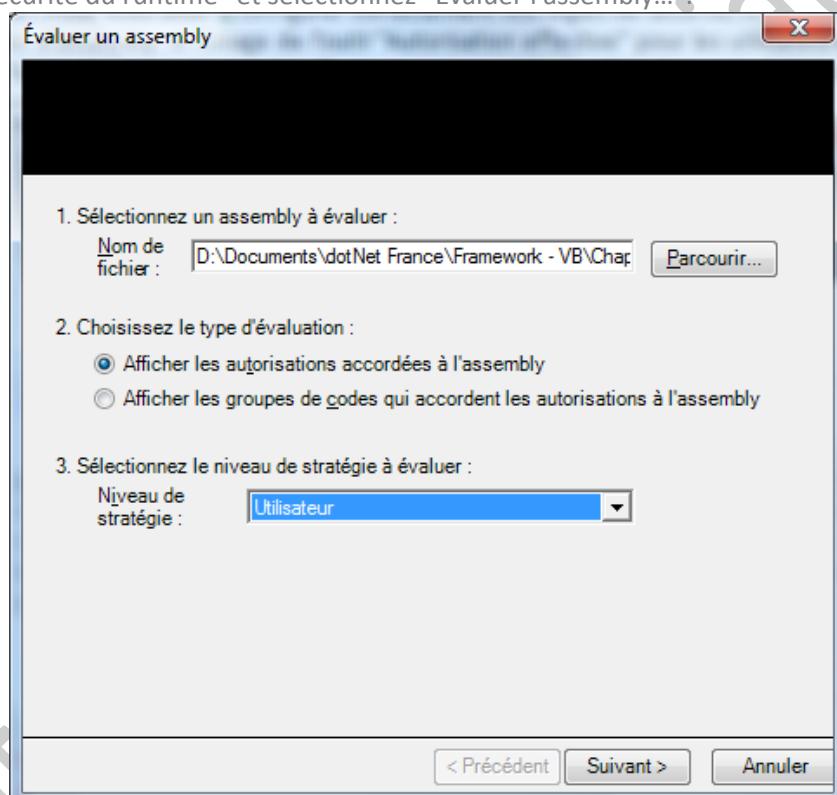


Malgré notre ligne d'affichage d'une boîte de dialogue SaveFileDialog, aucune autre fenêtre que la fenêtre principale ne sera affichée ; et c'est entièrement explicite. Rappelez-vous, dans la partie précédente, notre jeu d'autorisations autorise l'affichage des fenêtres (Interface utilisateur) mais interdit l'affichage des boîtes de dialogue en sauvegarde. Cette interdiction lève une exception affichée dans la console.

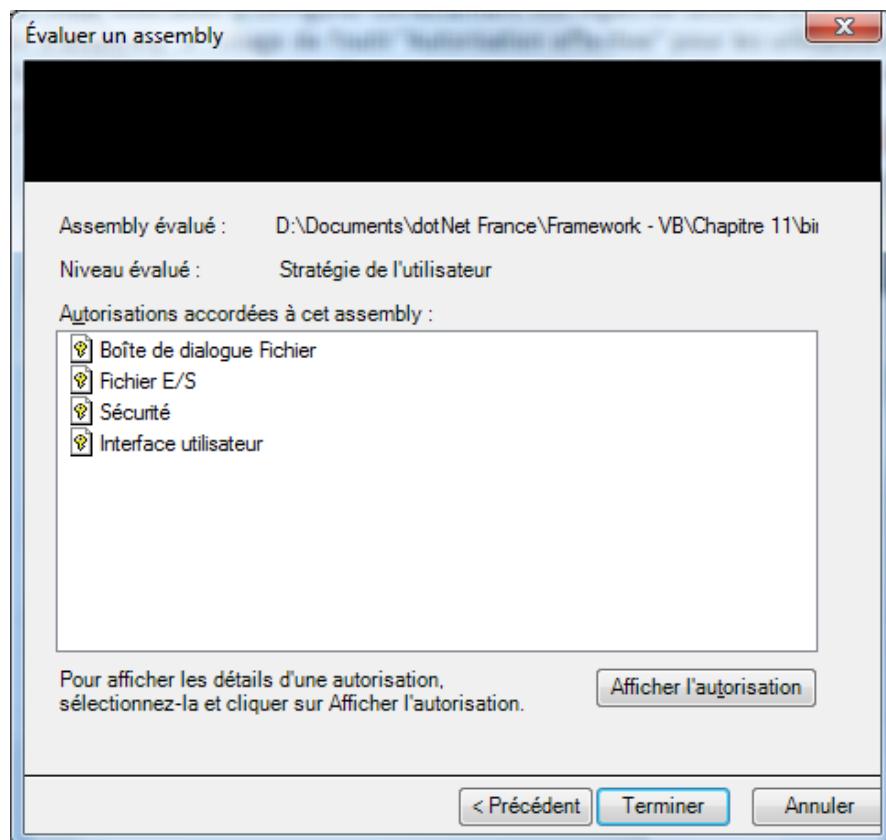


**Attention :** La sécurité du .NET Framework peut être tout aussi puissante que le système RBS pour les utilisateurs de la machine Windows. Seulement, ce qui fait sa puissance peut aussi rapidement devenir un véritable désastre si elle est mal configurée. Des règles de sécurité trop exigeantes peuvent même vous empêcher de lancer Visual Studio ; ce dernier utilisant certaines assemblies du Framework, il générera des erreurs à tour de bras, vous empêchant ainsi toute utilisation normale.

**Note :** Pour vous aider à configurer correctement vos règles de sécurité, n'hésitez pas utiliser l'évaluation d'assembly. A l'image de l'outil "Autorisation effective" pour les utilisateurs, il permet d'afficher toutes les règles de sécurité affectant une assembly spécifiée. Pour cela, faites clic-droit sur "Stratégie de sécurité du runtime" et sélectionnez "Evaluer l'assembly..."



Il ne vous reste plus qu'à sélectionner le fichier contenant l'assembly à évaluer, à indiquer quelles informations afficher et cliquer sur Suivant :



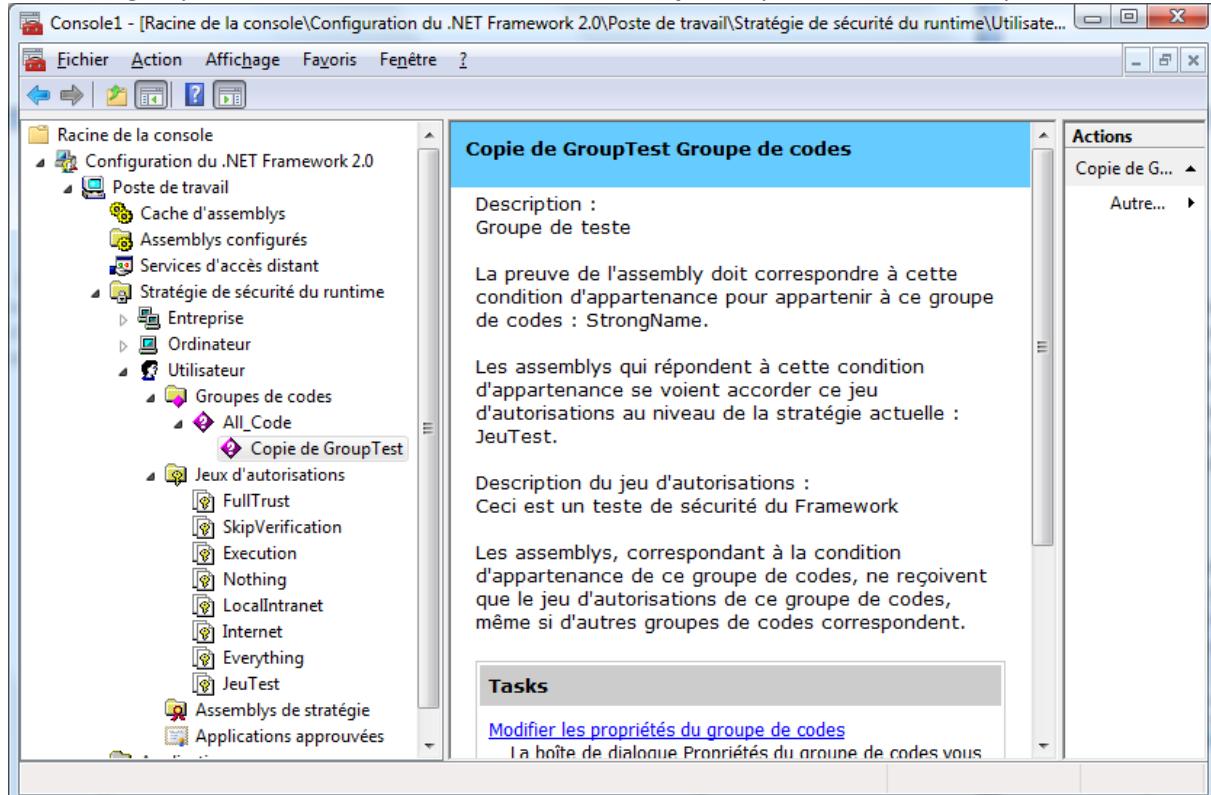


### 2.3.2.2 Notion de hiérarchie des groupes

Dans l'exemple plus haut, je vous avais demandé de cocher la case "Ce niveau de stratégie ne bénéficiera que des autorisations du jeu d'autorisation associé à ce groupe de codes" sans comprendre pourquoi. La raison est simple : les groupes de codes se basent sur un système hiérarchisation des droits.

Dans un code orienté objet, en général, quand une classe hérite d'une autre, la classe fille dispose de tous les membres de la classe parente.

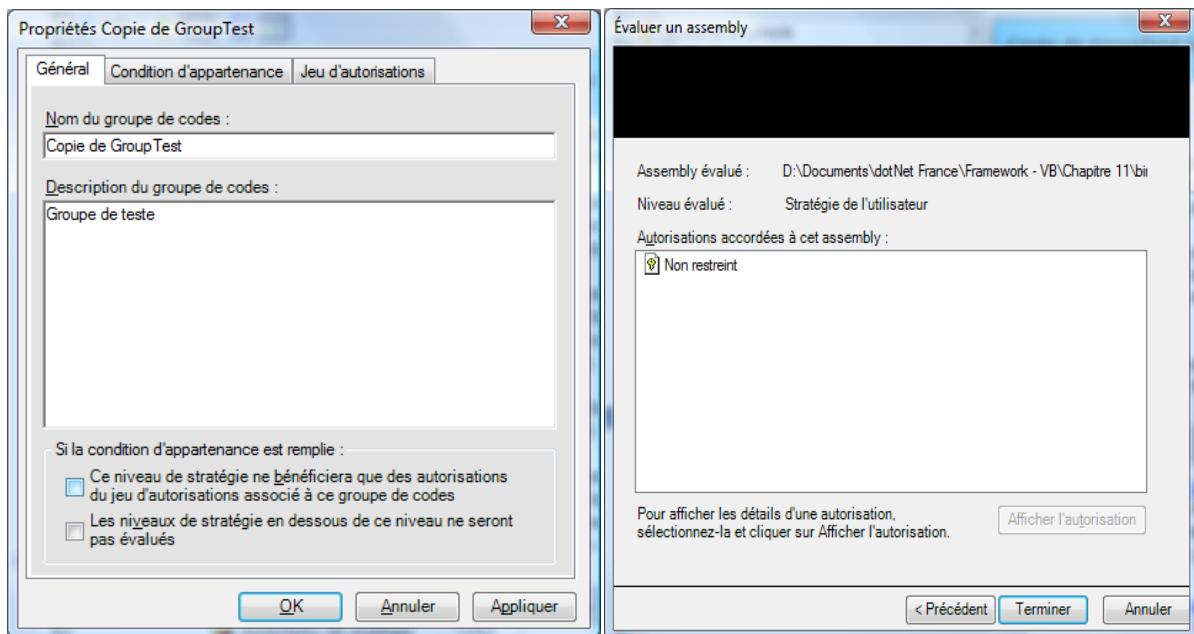
Les groupes de codes fonctionnent de la même façon. Reprenons notre exemple :



Ici nous pouvons voir clairement que "Copie de GroupTest" est un groupe de code enfant du groupe "All\_code".

Tout à l'heure, nous avions affiché les autorisations effectives sur l'assembly et avons constaté que les règles définies par notre jeu d'autorisation étaient bien présentes.

Maintenant, si nous modifions le groupe de code "Copie de GroupTest", que nous décochons la case indiquée et que nous affichons à nouveau les autorisations effectives :



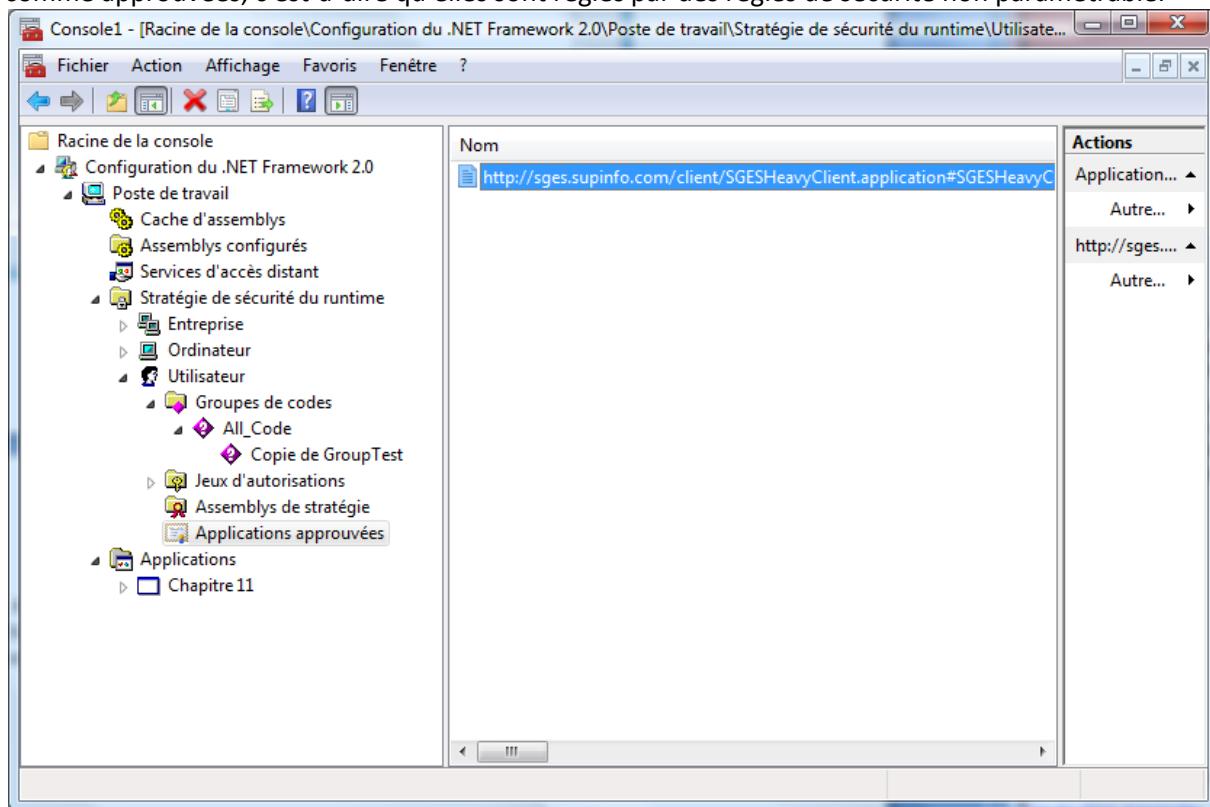
Nous pouvons remarquer à présent que l'assembly est passée en "Non restreint".

Cette case "Ce niveau [...]" permet en fait de forcer la CLR à appliquer uniquement les permissions du jeu d'autorisation lié au groupe de code et ce, sans tenir compte des autorisations du groupe de code parent. Si nous la décochons, notre groupe de code "Copie de GroupTest" va de nouveau hériter des règles de sécurité du groupe "All\_code" qui est configuré pour permettre à n'importe quel programme de gérer n'importe quelle ressource.



### 2.3.3 Applications approuvées

Cette section regroupe les applications "ClickOnce" (à déploiement rapide) considérées comme approuvées, c'est-à-dire qu'elles sont régies par des règles de sécurité non paramétrable.



### 2.3.4 Le CAS et les permissions d'hôte

Quand vous configurez les règles de sécurité du .NET Framework, il faut bien garder en tête qu'elles ne prévalent en rien sur les règles de sécurité établies par l'OS qui sera chargé d'exécuter votre assembly. Si, par exemple, le Framework est configuré pour avoir un accès total au disque C:\ mais que l'utilisateur ne possède pas ces droits au niveau du système d'exploitation, toute tentative d'écriture, de lecture ou d'exécution sur C:\ échouera.

## 2.4 Configuration de sécurité en ligne de commande

Pour les adeptes de la ligne de commande ou des scripts batch, il est tout à fait possible de configurer la sécurité du Framework avec une invite de commande. Pour cela, il vous suffit d'utiliser le logiciel "Caspol.exe" qui se trouve dans le dossier "C:\Windows\Microsoft.NET\Framework\v2.0.50727". Si vous souhaitez avoir plus d'informations sur l'utilisation de cet outil, je vous recommande [ce lien](#) qui explique tous les paramètres de cet exécutable.

### 3 Gestion de la sécurité du programme

Nous avons vu dans la première partie beaucoup de théorie sur la sécurité. A présent, nous allons voir comment faire en sorte que vos assemblies possèdent strictement les droits dont elles ont besoin afin de protéger le système sur lequel elles sont utilisées.

#### 3.1 Restrictions d'exécution au niveau de l'assembly

Pour cela nous allons indiquer explicitement, grâce à un ensemble de classes sous forme d'attributs et trois valeurs de l'énumération `SecurityAction`, les permissions nécessaires au bon fonctionnement de l'application.

Grâce à cette méthode nous pouvons nous assurer que notre assembly n'exécutera pas certains bouts de codes sans en avoir la permission. Le Runtime va d'abord vérifier que les demandes de permissions minimum peuvent être accordées et pourra renvoyer une exception si l'une d'entre elle est refusée. Cela permet également d'appliquer l'un des principes de base de la sécurité, à savoir qu'il faut toujours donner à une application ou un utilisateur seulement les droits nécessaires et suffisants. Pour illustrer ce principe, on parle souvent de liste blanche et liste noire : Plutôt que de faire une liste des interdits sur une liste noire, il vaut mieux faire une liste des autorisations sur une liste blanche.

Tout d'abord nous allons voir en détail les trois types de déclarations possibles. Celles-ci font partie de l'énumération `SecurityAction` et ne s'appliquent qu'à l'assembly. D'autres types de déclarations sont disponibles dans `SecurityAction` mais ne s'appliquent qu'aux classes et/ou méthodes :

Valeur	Description
<code>RequestMinimum</code>	Associé à une permission, permet d'indiquer que les demandes d'accès sont analysées avant l'exécution du code. Si une demande de permission est refusée, une exception est levée.
<code>RequestOptional</code>	Permet de créer une liste blanche des demandes de permissions. Ces demandes seront analysées au moment de l'exécution du code. A partir du moment où une demande d'accès est déclarée <code>RequestOptional</code> , toutes les autres demandes qui ne sont pas faites explicitement seront refusées et lèveront des exceptions.
<code>RequestRefuse</code>	Permet de refuser explicitement une permission de manière à créer une liste noire. Une demande de permission refusée lève une exception.

**Note** : Le nom `RequestOptional` peut porter à confusion. En effet il n'indique pas une demande de permission optionnelle mais est bien obligatoire pour autoriser des actions.

Dans chaque demande de permission, vous pouvez faire suivre votre paramètre `SecurityAction` par d'autres paramètres sous la forme d'une propriété suivis d'un égal et de sa valeur. Par exemple, la plus courante `Unrestricted` permet d'autoriser (true) ou non (false) l'accès complet à la ressource.

Nous verrons deux autres propriétés dans l'exemple de code.

Dans tous les cas, il est important de surveiller les exceptions de type `SecurityException` grâce à des blocs Try-Catch. Ainsi, si une demande de droit n'a pu aboutir, vous pourrez indiquer dans la console ou le journal d'évènements les permissions à activer pour que l'application fonctionne.



Maintenant que nous avons vu les trois principales valeurs qui nous intéressent dans l'énumération **SecurityAction**, nous allons détailler les demandes de permissions principales misent à disposition par le Framework .NET :

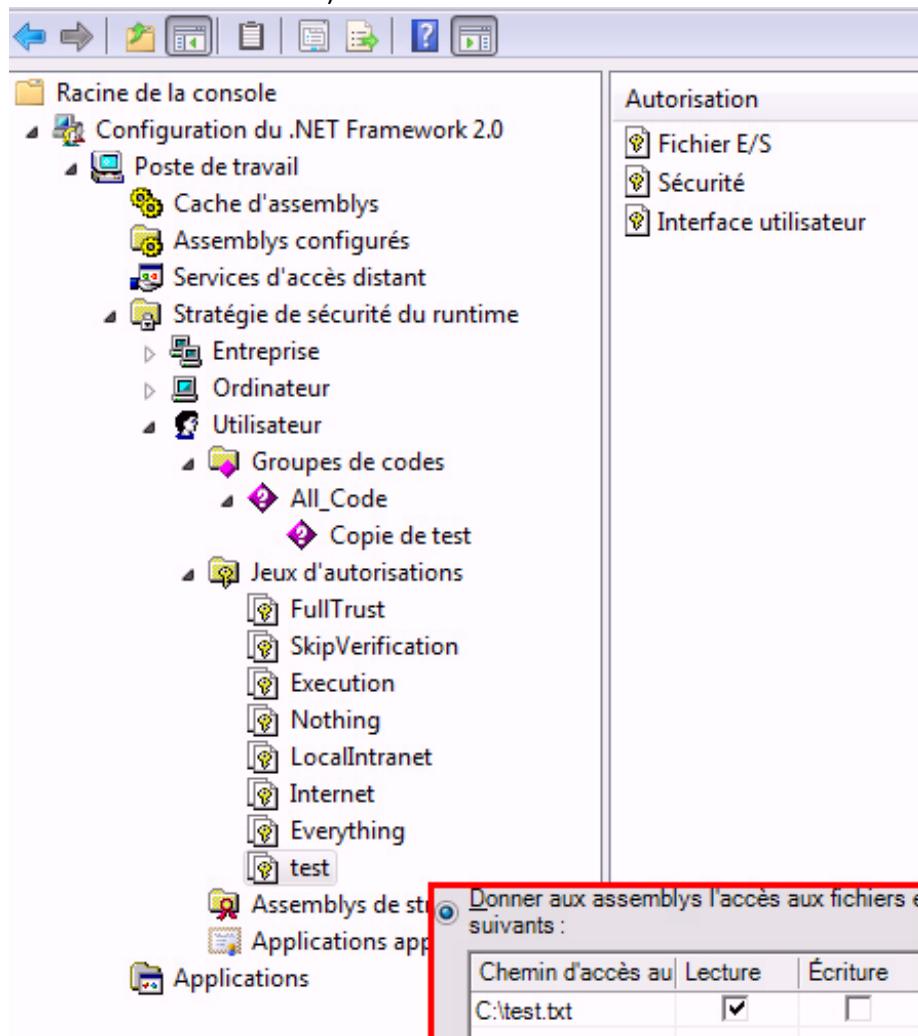
Classes	Description
<code>AspNetHostingPermission</code>	Demande d'accès entre le système et un environnement Asp.NET.
<code>DataProtectionPermission</code>	Demande d'accès à la mémoire et à des données chiffrées.
<code>DirectoryServicesPermission</code>	Demande d'accès à l'espace de nom <code>System.DirectoryServices</code> .
<code>DnsPermission</code>	Demande d'accès à un serveur DNS.
<code>EnvironmentPermission</code>	Demande d'accès aux variables du système et de l'environnement de l'utilisateur.
<code>EventLogPermission</code>	Demande d'accès au journal d'évènements.
<code>FileDialogPermission</code>	Demande d'accès à un fichier du système à partir d'une boîte de dialogue.
<code>FileIOPermission</code>	Demande d'accès à un fichier du système.
<code>IsolatedStorageFilePermission</code>	Demande d'accès aux zones de stockages isolées
<code>KeyContainerPermission</code>	Demande d'accès à un conteneur de clé. Voir la partie sur la sécurité des données.
<code>OdbcPermission</code>	Demande d'accès à une source de donnée ODBC.
<code>OleDbPermission</code>	Demande d'accès à une base de données en utilisant OleDb.
<code>OraclePermission</code>	Demande d'accès à une base de données Oracle.
<code>PerformanceCounterPermission</code>	Demande d'accès aux compteurs de performances.
<code>PrintingPermission</code>	Demande d'accès aux imprimantes.
<code>ReflectionPermissions</code>	Demande de permission d'utilisation des outils de l'espace de nom <code>System.Reflection</code> .
<code>RegistryPermission</code>	Demande d'accès au registre Windows.
<code>ServiceControllerPermission</code>	Demande d'accès au contrôle des services.
<code>SocketPermission</code>	Demande l'autorisation de créer une connexion par sockets.
<code>SqlClientPermission</code>	Demande l'accès à une base de données SQL.
<code>UIPermission</code>	Demande d'accès à l'interface utilisateur et au presse-papier. Nécessaire au débuggeur de Visual Studio.
<code>WebPermission</code>	Demande d'accès à une ressource sur internet par HTTP.

**Note :** Ces classes possèdent toutes une version attribut mais vous pouvez les utiliser en tant qu'objets. Nous ne détaillerons pas cette façon de faire.



Nous allons illustrer un peu ce que nous avons vu avec un court exemple :

Tout d'abord nous créons un fichier texte à la racine de notre disque C : que nous appelons test.txt. Ensuite, nous relançons la console MMC et nous autorisons l'accès au fichier par notre assembly uniquement en lecture (Par souci de simplicité, ajoutez également l'accès total à la gestion de la sécurité et à l'interface utilisateur).



Ensuite nous allons accéder dans notre application une première fois en lecture et une seconde en écriture.



```
' VB
Imports System.IO
Imports System.Security.Permissions

<Assembly: UIPermission(SecurityAction.RequestOptional,
Clipboard:=UIPermissionClipboard.AllClipboard,
Window:=UIPermissionWindow.AllWindows)> _
<Assembly: FileIOPermission(SecurityAction.RequestMinimum,
Read:="C:\test.txt")> _
<Assembly: FileIOPermission(SecurityAction.RequestOptional,
Write:="C:\test.txt")> _
Module partie2
    Public Sub Main()
        Try
            Console.WriteLine("Tentative de lecture")
            Dim sr As StreamReader = New StreamReader("c:\test.txt")
            sr.Close()
            Console.WriteLine("Tentative d'écriture")
            Dim sw As StreamWriter = New StreamWriter("c:\test.txt")
            sw.Close()
        Catch ex As Exception
            Console.WriteLine(ex.Message)
        End Try
        Console.Read()
    End Sub
End Module
```

```
//C#
using System;
using System.IO;
using System.Security.Permissions;

[assembly: UIPermission(SecurityAction.RequestOptional, Window =
UIPermissionWindow.AllWindows, Clipboard =
UIPermissionClipboard.AllClipboard)]
[assembly: FileIOPermission(SecurityAction.RequestMinimum, Read =
@"C:\test.txt")]
[assembly: FileIOPermission(SecurityAction.RequestOptional, Write =
@"C:\test.txt")]
namespace Chapitre_11
{
    class Partie2
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Tentative de lecture")
                StreamReader sr = new StreamReader(@"c:\test.txt");
                sr.Close();
                Console.WriteLine("Tentative d'écriture")
                StreamWriter sw = new StreamWriter(@"c:\test.txt");
                sw.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.Read();
        }
    }
}
```

Nous utilisons ici quatre attributs :

- **UIPermission** est en `RequestOptional`. Cela veut dire que dans toute l'assembly, les demandes d'accès à cette ressource seront testées après évaluation du code. Cela veut également dire que dans toutes mon assembly, toute demande de permission non explicitée sera refusée. Nous lui avons assigné deux propriétés permettant d'indiquer que nous demandons l'accès à toutes les fenêtres et également à tous les presse-papiers. Cet attribut est absolument nécessaire si vous exécutez votre application en utilisant le débuggeur Visual Studio.
- Le premier **FileIOPermission** nous permet de demander la permission d'accéder en lecture au fichier test.txt. C'est une demande minimum, cela veut dire que si elle n'est pas acceptée, une exception sera levée avant même que la tentative d'accès au fichier soit effectuée.
- Le second **FileIOPermission** nous permet de demander la permission d'accéder en écriture au fichier test.txt. C'est une demande optional c'est-à-dire que l'analyse des règles de sécurité sera faite au moment où une demande d'accès en écriture au fichier est effectuée. Si elle est refusée, une exception sera levée.



Si nous lançons l'application, voilà l'exception qui sera levée :

Tentative de lecture

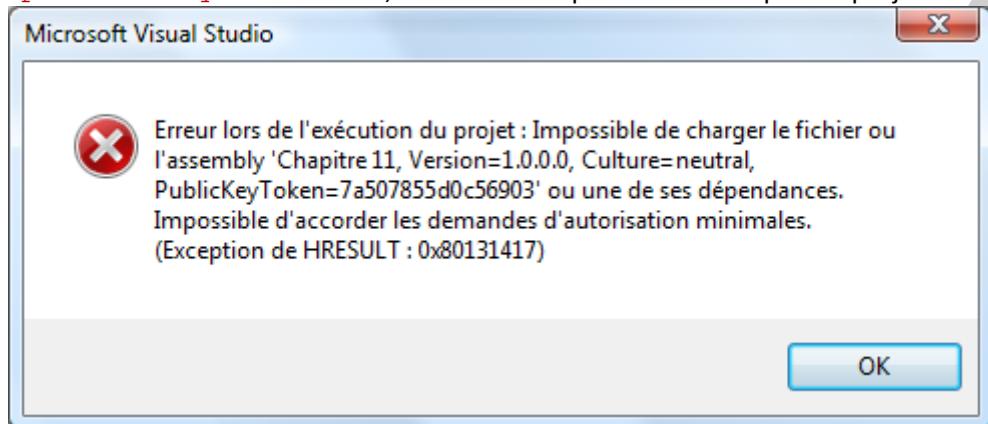
Tentative d'écriture

Échec de la demande d'autorisation de type 'System.Security.Permissions.FileIOPermission, mscorelib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.

Il en sera de même si vous ne laissez que le premier FileIOPermission.

L'exception nous indique que nous n'avons pas l'accès à test.txt en écriture ; La demande est donc refusée.

En revanche, si nous changeons le second paramétrage FileIOPermission de **RequestOptional** à **RequestMinimum**, il nous sera impossible de compiler le projet :



Pour finir, supprimez les règles de sécurité définie dans ce test grâce à la console MMC et modifiez le second paramétrage FileIOPermission de **RequestOptional** à **RequestRefuse**.

Tentative de lecture

Tentative d'écriture

Échec de la demande d'autorisation de type 'System.Security.Permissions.FileIOPermission, mscorelib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.

Nous aurons droit à la même erreur que si nous avions laissé le paramétrage de la CLR avec une demande **RequestOptional**.

Maintenant que nous savons comment restreindre les permissions au niveau de l'assembly, nous pourrions également nous demander comment faire pour appeler du code de confiance totale à partir d'une assembly de confiance partielle. Pour cela, rien de plus simple que d'ajouter l'attribut suivant :

```
'VB
<Assembly: System.Security.AllowPartiallyTrustedCallers()>

//C#
[assembly: System.Security.AllowPartiallyTrustedCallers]
```

Sachez toutefois que les membres publics d'une assembly non-signée sont accessibles depuis l'extérieur, même si cet attribut n'est pas présent !



### 3.2 Modification des droits au niveau d'une classe/méthode

Nous avons vu précédemment comment appliquer des mesures de sécurité à une assembly par le code, nous allons maintenant voir comment les appliquer directement à des classes ou des méthodes.

Nous avions vu dans la partie précédente l'utilisation des attributs afin de créer des demandes de permissions déclaratives en utilisant trois valeurs de l'énumération **SecurityAction**. Au niveau des méthodes et des classes, nous avons accès à de nouvelles valeurs que voici :

Valeur	Description
<b>Assert</b>	Permet au code associé d'être exécuté par d'autres portions de code se trouvant après (dans la pile d'appel) la partie équipée d'une demande de permission Assert et ce, même s'ils n'en font pas la demande. Bien entendu la permission doit être accordée dans l'assembly contenant le code. La permission autorisée sera automatiquement restaurée à son état d'origine lorsque la pile d'appel sera parcourue en sens inverse.
<b>Demand</b>	Fournit, à la différence de Assert, une permission à une seule portion de code. Tous les autres codes se trouvant avant ou après dans la pile d'appel doivent également faire une demande pour disposer des mêmes droits.
<b>Deny</b>	Permet de refuser explicitement un droit à une portion de code. Tout code se situant après dans la pile d'appel aura également une interdiction de droit, sauf s'il fait une demande de droits.
<b>InheritanceDemand</b>	Permet de s'assurer que les classes qui héritent ou qui substituent une méthode de la classe associée ont les autorisations nécessaires.
<b>LinkDemand</b>	Fourni une permission à une portion de code ainsi qu'à tout code se trouvant après dans la pile d'appel. A la différence d'Assert, la permission ne sera restaurée lors de la remontée de la pile d'appel ce qui peut représenter une importante faille de sécurité.
<b>PermitOnly</b>	Supprime toute autorisation accordée jusqu'ici et autorise uniquement celle qui est affectée de PermitOnly.

Note : La notion de pile d'appel est expliquée [ici](#).

Comme ces notions sont un peu délicates à comprendre, nous allons essayer de les comprendre en prenant un exemple de la vie courante. Imaginez que vous travaillez dans la S.A Dotnet-France. Les locaux sont équipés d'une porte d'entrée qu'il faut sécuriser afin de protéger l'accès.

Nous avons plusieurs choix qui s'offrent à nous pour sécuriser l'entrée. A chaque valeur de l'énumération précédente, nous associons un dispositif particulier :

- **Assert** peut être comparé à un digicode. Dès qu'une personne entre le bon code, on lui donne la possibilité de laisser entrer autant de personnes qu'elle le souhaite ; Cependant, nous obligerons la personne à refermer la porte. Nous faisons alors confiance en la personne qui a tapé le code pour qu'elle ne soit accompagnée que de personnes autorisées. On gagne ainsi du temps, mais en contrepartie, si un intrus se glisse dans le groupe nous nous exposons à un risque.
- **Demand** peut être associé à un dispositif d'ouverture par badge. Chaque employé doit ouvrir l'entrée grâce à son badge, et nous savons ainsi qui rentre et à quelle heure.
- **Deny** et **PermitOnly** peuvent être comparé à un vigile qui va dans le premier cas empêcher certaines personnes d'entrer sur un critère en particulier, il va par exemple interdire l'entrée à tous ceux qui sont en Tee-shirt. Dans l'autre cas, le vigile va seulement autoriser les personnes correspondant à certains critères, par exemple il laissera entrer seulement les personnes en costume.



- Dans le cas de **LinkDemand**, nous avons choisi simplement de fermer la porte à clef, dès qu'une personne a ouvert la porte, tous le monde peut entrer, tous le processus est simplifié, mais les intrus peuvent rentrer très facilement.
- **InheritanceDemand** est une version plus poussée du dispositif d'ouverture par badge. On peut imaginer que nous avons rajouté un vigile qui va vérifier que les personnes qui accompagnent un employé sont autorisées à entrer (par exemple des clients).

Maintenant que nous y voyons un peu plus clair sur la méthode déclarative de demande de permissions, nous allons parler un petit peu de la méthode impérative.

Nous avons vu que pour faire une demande de permission déclarative, il nous fallait utiliser un attribut au dessus de nos classes ou méthodes. Pour faire une demande de permission impérative, nous allons instancier un objet qui hérite de **CodeAccessPermission** et utiliser les méthodes qui conviennent. Cela va nous permettre de faire des demandes de permissions pour une partie précise d'un bloc plutôt que pour le bloc entier.

Les méthodes de **CodeAccessPermission** ne vous seront pas étrangères puisqu'elles portent le même nom que les valeurs de l'énumération **SecurityAction** et ont les mêmes comportements. En revanche **LinkDemand** et **InheritanceDemand** n'existent pas.

Afin de comprendre la différence entre les deux nous allons voir un exemple d'utilisation des deux méthodes. Notre exemple ne va utiliser que **Demand**, les autres actions seront vues ensuite.

```
'VB
Imports System.IO
Imports System.Security.Permissions
<FileIOPermission(SecurityAction.Demand, Write:="C:\")> _
Public Sub Ecrire()
    Dim sw As StreamWriter = New StreamWriter("C:\test.txt")
    sw.WriteLine("blabla")
    sw.Flush()
    sw.Close()
End Sub
Public Sub EcrireDeux()
    Dim FP As FileIOPermission = New
FileIOPermission(FileIOPermissionAccess.Write, "C:\test.txt")
    FP.Demand()
    Dim sw As StreamWriter = New StreamWriter("C:\test.txt")
    sw.WriteLine("tructruc")
    sw.Flush()
    sw.Close()
End Sub
Public Sub Lire()
    Dim sr As StreamReader = New StreamReader("C:\test.txt")
    Console.WriteLine(sr.ReadToEnd())
    sr.Close()
End Sub
Public Sub Main()
    Ecrire()
    Lire()
    EcrireDeux()
    Lire()
    Console.Read()
End Sub
```

```
//C#
using System.Security.Permissions;
using System.IO;
[FileIOPermission(SecurityAction.Demand, Write = @"C:\")]
public static void Ecrire()
{
    StreamWriter sw = new StreamWriter(@"C:\test.txt");
    sw.WriteLine("blabla");
    sw.Flush();
    sw.Close();
}
public static void EcrireDeux()
{
    FileIOPermission FP = new
    FileIOPermission(FileIOPermissionAccess.Write, @"C:\test.txt");
    FP.Demand();
    StreamWriter sw = new StreamWriter(@"C:\test.txt");
    sw.WriteLine("tructruc");
    sw.Flush();
    sw.Close();
}
public static void Lire()
{
    StreamReader sr = new StreamReader(@"C:\test.txt");
    Console.WriteLine(sr.ReadToEnd());
    sr.Close();
}
public static void Main(string[] args)
{
    Ecrire();
    Lire();
    EcrireDeux();
    Lire();
    Console.Read();
}
```

Nous avons donc utilisé la méthode déclarative pour la première méthode `Ecrire`. Nous demandons l'autorisation d'écrire sur tout le lecteur C, si la permission est accordée, nous pourrons écrire partout sur ce lecteur. Cependant, vous pouvez avoir les droits d'accès au lecteur C sans pour autant les avoir sur le fichier concerné. Dans tous les cas, si vous ne possédez pas les droits nécessaires, une exception sera levée.

Dans la deuxième méthode `EcrireDeux`, nous faisons une demande de permission impérative. Nous indiquons que nous voulons écrire sur le fichier test.txt en particulier. Grâce à cette méthode nous allons pouvoir gérer les cas où une demande est refusée afin qu'aucune exception ne soit levée, nous allons le voir dans un exemple à venir.

Bien entendu, les deux méthodes `Ecrire` et `EcrireDeux` ont le même comportement final comme en atteste la console :

blabla

tructruc

Nous allons reprendre notre exemple précédent et gérer le cas où la demande serait refusée. Par exemple nous pourrons afficher une boîte de dialogue demandant à quel endroit l'application pourra enregistrer le fichier. Vous aurez besoin d'importer l'espace de nom `System.Security` pour effectuer la vérification :

```
'VB
Imports System.IO
Imports System.Security.Permissions
Imports System.Security
Public Sub EcrireDeux()
    Dim FP As FileIOPermission = New
FileIOPermission(FileIOPermissionAccess.Write, "C:\test.txt")
    If (SecurityManager.IsGranted(FP)) Then
        Dim sw As StreamWriter = New StreamWriter("C:\test.txt")
        sw.WriteLine("tructruc")
        sw.Flush()
        sw.Close()
    Else
        Console.WriteLine("Erreur")
    End If
End Sub

//C#
using System.Security.Permissions;
using System.IO;
using System.Security;
public static void EcrireDeux()
{
    FileIOPermission FP = new
FileIOPermission(FileIOPermissionAccess.Write, @"C:\test.txt");
    if (SecurityManager.IsGranted(FP) == true)
    {
        StreamWriter sw = new StreamWriter(@"C:\test.txt");
        sw.WriteLine("tructruc");
        sw.Flush();
        sw.Close();
    }
    else
    {
        Console.WriteLine("Erreur");
    }
}
```

Nous avons un peu modifié notre méthode `EcrireDeux`, tout d'abord, nous n'utilisons plus les méthodes de `CodeAccessPermission`. Nous instancions juste un objet `FileIOPermission` qui sera vérifié dans la condition suivante. Nous vérifions que la permission est accordée grâce à la méthode statique `SecurityManager.IsGranted`. Si la valeur renournée est true, alors c'est que la permission est accordée, nous pouvons écrire dans notre fichier.

Bien entendu, s'il est possible de faire des demandes de permissions, il est également possible de les interdire. Pour cela, vous pouvez procéder également de façon impérative ou déclarative.

Pour refuser des droits explicitement, vous pouvez utiliser :

- **Deny** : Cette valeur va nous permettre de supprimer une permission ou un jeu d'autorisations de la méthode concernée (Elle va créer une liste noire des permissions effectives sur la méthode).
- **PermitOnly** : Permet de supprimer toutes les autorisations accordées jusqu'ici et de n'autoriser que celles qui sont affectées par ce paramètre (Elle va créer une liste blanche des permissions effectives sur la méthode).

Dans cet exemple, nous créons plusieurs méthodes permettant d'accéder au fichier C:\test.txt ainsi qu'une méthode lançant les deux autres mais qui est équipée d'une restriction déclarative :



```
' VB
Imports System.IO
Imports System.Security.Permissions
Public Sub ReadTest()
    Dim sr As StreamReader = New StreamReader("c:\test.txt")
    sr.Close()
End Sub
Public Sub WriteTest()
    Dim sw As StreamWriter = New StreamWriter("c:\test.txt")
    sw.Close()
End Sub

<FileIOPermission(SecurityAction.PermitOnly, Read:="C:\test.txt")> _
Public Sub DoneAll()
    ReadTest()
    WriteTest()
End Sub
Public Sub Main()
    Try
        Console.WriteLine("Tentative de lecture")
        ReadTest()
        Console.WriteLine("Tentative d'écriture")
        WriteTest()
        Console.WriteLine("Opération groupée")
        DoneAll()
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
    Console.Read()
End Sub
```

```
//C#
using System.IO;
using System.Security.Permissions;
public static void ReadTest()
{
    StreamReader sr = new StreamReader(@"C:\test.txt");
    sr.Close();
}
public static void WriteTest()
{
    StreamWriter sr = new StreamWriter(@"C:\test.txt");
    sr.Close();
}
[FileIOPermission(SecurityAction.PermitOnly, Read="C:\test.txt")]
public static void DoneAll()
{
    ReadTest();
    WriteTest();
}
public static void Main(string[] args)
{
    try
    {
        Console.WriteLine("Tentative de lecture");
        ReadTest();
        Console.WriteLine("Tentative d'écriture");
        WriteTest();
        Console.WriteLine("Opération groupée");
        DoneAll();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.Read();
}
```

Tentative de lecture

Tentative d'écriture

Opération groupée

Échec de la demande d'autorisation de type 'System.Security.Permissions.FileIOPermission, mscorelib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.

Nous constatons qu'aucune exception n'est levée si nous passons par l'une des méthodes ReadTest ou WriteTest. En revanche, si nous appelons la méthode DoneAll, l'accès en écriture échouera.

Le résultat aurait été strictement identique si nous avions remplacé "SecurityAction.PermitOnly, Read" par "SecurityAction.Deny, Write ". La seule différence est que dans ce cas-ci, nous n'autorisons que la lecture et dans l'autre, nous interdisons l'écriture.

Nous pouvons également effectuer les mêmes opérations de façon impérative :

```
'VB
Dim restrict As FileIOPermission
Public Sub Main()
    restrict = New FileIOPermission(FileIOPermissionAccess.Write,
"C:\test.txt")
    Try
        restrict.Demand()
        Console.WriteLine("Tentative de lecture")
        ReadTest()
        Console.WriteLine("Tentative d'écriture")
        WriteTest()
        restrict.Deny()
        Console.WriteLine("Opération groupée")
        DoneAll()
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
    Console.Read()
End Sub

//C#
private static FileIOPermission restrict;
public static void Main(string[] args)
{
    restrict = new FileIOPermission(FileIOPermissionAccess.Write,
"C:\test.txt");
    try
    {
        restrict.Demand();
        Console.WriteLine("Tentative de lecture");
        ReadTest();
        Console.WriteLine("Tentative d'écriture");
        WriteTest();
        restrict.Deny();
        Console.WriteLine("Opération groupée");
        DoneAll();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.Read();
}
```

Ce code donnera le même résultat que ci-dessus.

**Important :** Lorsque vous traitez vos exceptions, préférez utiliser la valeur `PermitOnly`. En effet, celle-ci est par définition beaucoup plus restrictive que `Deny`. Cela permet d'éviter qu'un pirate informatique utilise vos gestions d'erreurs afin de provoquer des comportements inhabituels de votre application.

Il se peut que vous ayez besoin de restreindre un ensemble de permissions dans une méthode. Vous allez pouvoir utiliser la classe `PermissionSet`. Celle-ci va vous permettre de créer un groupe de permissions représentée par un seul objet.

Voici un exemple :

```
//C#
Imports System.IO
Imports System.Security.Permissions
Imports System.Security
Dim permissions As PermissionSet = New
PermissionSet(PermissionState.None)
permissions.AddPermission(New
FileIOPermission(FileIOPermissionAccess.Write, "C:\test.txt"))
permissions.AddPermission(New
FileIOPermission(FileIOPermissionAccess.Write, "C:\test1.txt"))
permissions.Demand()
```

```
//C#
using System.Security.Permissions;
using System.IO;
using System.Security;
PermissionSet permissions = new PermissionSet(PermissionState.None);
permissions.AddPermission(new
FileIOPermission(FileIOPermissionAccess.Write, @"C:\test.txt"));
permissions.AddPermission(new
FileIOPermission(FileIOPermissionAccess.Write, @"C:\test1.txt"));
permissions.Demand();
```

Il peut être intéressant également de transformer l'objet PermissionSet en XML pour pouvoir l'importer ensuite dans d'autres méthodes. Pour cela vous devrez utiliser la méthode ToXml. Voici ce qui ressort si nous convertissons le PermissionSet ci-dessus en XML :

```
<PermissionSet class="System.Security.PermissionSet" version="1">
<IPermission class="System.Security.Permissions.FileIOPermission, mscorel, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" version="1" Write="C:\test.txt;C:\test1.txt"/>
</PermissionSet>
```

**Note** : Toute classe basée sur CodeAccessPermission possède également les méthodes ToXml et FromXml.

### 3.3 Effectuer des demandes groupées

Pour des raisons évidentes de performance de votre programme, il est possible d'éviter de faire des demandes de droits à chaque fois que nous en avons besoin. Pour cela, vous pouvez faire de deux façons :

- En utilisant LinkDemand. Cette méthode est pratique mais peut constituer un trou de sécurité important. En effet, si les autorisations sont accordées dans le sens descendant de la pile d'appel, à partir du moment où une ligne possède un attribut LinkDemand, les autorisations ne seront plus vérifiées au moment où la pile d'appel est vidée ; Seulement, les anciens droits d'accès ne pourront pas être restaurés à leur état d'origine. Ainsi, un code malicieux se trouvant tout en haut de la pile d'appel pourrait profiter des droits accordés par la méthode équipée de LinkDemand afin d'exécuter du code non-désirable.
- En utilisant Assert. Cette méthode est similaire à LinkDemand à la nuance près que lorsque l'autorisation a été accordée, elle est supprimée par la suite. Ainsi, là où LinkDemand laissera des droits à l'application, Assert restaurera les restrictions d'accès. Une méthode se trouvant en haut de la pile d'appel ne pourra plus exécuter de code en profitant des droits accordés temporairement.



**Note :** Aucun exemple ne peut être facilement fourni avec LinkDemand. En effet, LinkDemand est plus complexe à illustrer et il agit au moment de la compilation JIT plutôt qu'à l'exécution du programme.

Voici un exemple avec Assert dans lequel nous appelons 2 méthodes qui tentent une écriture sur un fichier dont l'accès est interdit en écriture :

```
'VB
<FileIOPermission(SecurityAction.Deny, Write:="C:\test.txt")> _
Public Sub Main()
    Try
        Console.WriteLine("Tentative d'opération groupée")
        DoneAll()
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try

    Try
        Console.WriteLine("Ecriture a partir de Main")
        Dim sw As StreamWriter = New StreamWriter("c:\test.txt")
        sw.Close()
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
    Console.Read()
End Sub
Public Sub WriteTest()
    Dim sw As StreamWriter = New StreamWriter("c:\test.txt")
    sw.Close()
End Sub

<FileIOPermission(SecurityAction.Assert, Write:="C:\test.txt")> _
Public Sub DoneAll()
    Dim sw As StreamWriter = New StreamWriter("c:\test.txt")
    sw.Close()
    WriteTest()
End Sub
```

Dotnet://



```
//C#
[FileIOPermission(SecurityAction.Deny, Write="C:\test.txt")]
public static void Main(string[] args)
{
    try {
        Console.WriteLine("Tentative d'opération groupée");
        DoneAll();
    }
    catch (Exception ex) {
        Console.WriteLine(ex.Message);
    }
    try {
        Console.WriteLine("Ecriture a partir de Main");
        StreamWriter sw = new StreamWriter(@"C:\test.txt");
        sw.Close();
    }
    catch (Exception ex) {
        Console.WriteLine(ex.Message);
    }

    Console.Read();
}
public static void WriteTest()
{
    StreamWriter sr = new StreamWriter(@"C:\test.txt");
    sr.Close();
}
[FileIOPermission(SecurityAction.Assert, Write="C:\test.txt")]
public static void DoneAll()
{
    StreamWriter sw = new StreamWriter(@"C:\test.txt");
    sw.Close();
    WriteTest();
}
```

Si nous compilons ce projet, nous obtenons ce résultat :

Tentative d'opération groupée  
 Ecriture a partir de Main  
 Échec de la demande d'autorisation de type 'System.Security.Permissions.FileIOPermission, mscorelib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.

Dans un premier temps, nous appelons la méthode DoneAll qui est équipée de l'autorisation en écriture avec Assert. Cette autorisation est effective sur la méthode et sur toutes les méthodes qui se trouvent après dans la pile d'appel, ce qui ne lèvera aucune exception.

Par contre, si nous tentons d'ouvrir le fichier en écriture dans la méthode Main, après l'appel à la méthode DoneAll, nous avons une exception de sécurité qui est levée.

Afin de vous prouver que les permissions sont analysées en fonction de la pile d'appel, reprenez l'exemple et supprimez l'attribut placé au dessus de la méthode DoneAll. Cette fois-ci, même à l'appel de la méthode, vous aurez une exception qui sera levée :

Tentative d'opération groupée  
 Échec de la demande d'autorisation de type 'System.Security.Permissions.FileIOPermission, mscorelib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.  
 Ecriture a partir de Main  
 Échec de la demande d'autorisation de type 'System.Security.Permissions.FileIOPermission, mscorelib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.



## 4 Sécurisation des accès

Nous avons vu dans une première partie comment sécuriser nos applications, nous allons maintenant voir comment protéger les données et les utilisateurs de vos applications.

Deux concepts de base à connaître quand on parle de sécurité des données et des utilisateurs sont l'authentification et l'autorisation :

- L'authentification est le fait de vérifier l'identité d'un utilisateur. Par exemple en lui demandant son login et mot de passe et en vérifiant que les bonnes données ont été entrées.
- L'autorisation permet de vérifier qu'un utilisateur, identifié ou non, a le droit d'accéder à certaines ressources.

Généralement, authentification et autorisation sont jumelées afin de maximiser la sécurité d'un système.





## 4.1 Authentification

Le Framework .NET propose deux classes permettant de représenter un utilisateur ou un groupe d'utilisateur. Grâce à ces deux classes nous allons pouvoir couvrir les besoins basiques de l'authentification. Ces deux classes se trouvent dans l'espace de nom `System.Security.Principal`.

**Note :** Ces deux classes ne fonctionnent que sous un environnement Windows.

### 4.1.1 Authentification avec `WindowsIdentity`

La classe `WindowsIdentity` permet de représenter un compte utilisateur sur les systèmes d'exploitation Windows. Vous ne pouvez pas authentifier un utilisateur auprès de votre application avec `WindowsIdentity` mais vous allez pouvoir récupérer, à partir du compte utilisateur authentifié, le nom d'un utilisateur présent dans la base de données [Active Directory](#), un jeton d'authentification et le résultat de l'authentification.

Voici les principaux membres de la classe `WindowsIdentity` :

Membres	Description
<code>GetAnonymous</code>	Méthode Statique, elle retourne un objet <code>WindowsIdentity</code> qui représente un utilisateur anonyme.
<code>GetCurrent</code>	Méthode Statique, elle retourne un objet <code>WindowsIdentity</code> qui représente l'utilisateur courant.
<code>Impersonate</code>	Impersonate possède une surcharge statique et une autre non statique : <ul style="list-style-type: none"> <li>➤ La méthode statique prend un <code>IntPtr</code> en paramètre représentant le jeton d'un utilisateur. Retourne un objet <code>WindowsIdentity</code> possédant l'identité de l'utilisateur représenté par le jeton.</li> <li>➤ La méthode non statique permet d'emprunter l'identité de l'utilisateur représenté par l'objet <code>WindowsIdentity</code></li> </ul>
<code>AuthenticationType</code>	Retourne le type d'authentification utilisé pour identifier l'utilisateur.
<code>Groups</code>	Retourne les groupes auxquels l'utilisateur appartient
<code>IsAnonymous</code>	Indique si l'utilisateur est identifié comme anonyme.
<code>IsAuthenticated</code>	Indique si l'utilisateur a été authentifié par Windows.
<code>IsGuest</code>	Indique si l'utilisateur a été authentifié comme compte Guest par Windows.
<code>IsSystem</code>	Indique si l'utilisateur a été authentifié comme compte System par Windows.
<code>Name</code>	Obtient le nom d'utilisateur utilisé lors de sa connexion. Le nom est composé du nom du Domaine puis du nom d'utilisateur, exemple : SCHMAC\Paul
<code>Owner</code>	Obtient l'identificateur de sécurité (SID) de l'utilisateur propriétaire du jeton.
<code>Token</code>	Obtient le jeton du compte utilisateur.
<code>User</code>	Obtient l'identificateur de sécurité (SID) de l'utilisateur.

Voici un exemple d'utilisation très simple de `WindowsIdentity`. Nous récupérons l'identité du compte utilisateur courant et affichons son nom d'utilisateur avec la propriété `Name` :

```
'VB
Imports System.Security.Principal
Sub Main()
    Dim IdentiteUtilisateur As WindowsIdentity =
    WindowsIdentity.GetCurrent()
    Console.WriteLine("Nom : " + IdentiteUtilisateur.Name)
    Console.Read()
End Sub

//C#
using System.Security.Principal;
static void Main()
{
    WindowsIdentity IdentiteUtilisateur = WindowsIdentity.GetCurrent();
    Console.WriteLine("Nom : " + IdentiteUtilisateur.Name);
    Console.Read();
}
```

Nom : SCHMAC\Paul

#### 4.1.2 Création d'une classe d'identité personnalisée

Nous avons vu juste avant la classe `WindowsIdentity` qui nous permet de représenter un utilisateur du Système sous forme d'un objet. Cette classe built-in dérive de l'interface `IIdentity`. Afin de créer une classe d'identité personnalisée, vous devrez implémenter l'interface `IIdentity`.

Voici les trois propriétés que vous devrez implémenter obligatoirement :

- `AuthenticationType` : C'est une chaîne de caractère (string) qui indique quel type d'authentification a été utilisée pour authentifier l'utilisateur.
- `IsAuthenticated` : C'est un booléen indiquant si l'utilisateur est authentifié ou non.
- `Name` : C'est une chaîne de caractère représentant le nom de l'utilisateur.

Ces propriétés étant par convention en lecture seule, vous devrez assigner vos valeurs soit par le constructeur de la classe, soit par une méthode.

Vous aurez sans doute besoin de créer vos propres classes d'identification si vous souhaitez utiliser un service autre que ceux dont le support est fourni par le Framework 2.0 (Une base de données, un logiciel de gestion à distance, etc...).

#### 4.1.3 Authentification d'un groupe d'utilisateurs

Pour authentifier un utilisateur selon son groupe d'utilisateur, nous allons utiliser la classe `WindowsPrincipal`. Pour récupérer l'objet `WindowsPrincipal` correspondant à l'identité de l'utilisateur, nous devons passer l'objet `WindowsIdentity` au constructeur de `WindowsPrincipal` :

```
'VB
Sub Main()
    Dim IdentiteUtilisateur As WindowsIdentity =
    WindowsIdentity.GetCurrent()
    Dim GroupeUtilisateur = New WindowsPrincipal(IdentiteUtilisateur)
End Sub
```



```
//C#
static void Main()
{
    WindowsIdentity IdentiteUtilisateur = WindowsIdentity.GetCurrent();
    WindowsPrincipal GroupeUtilisateur = new
WindowsPrincipal(IdentiteUtilisateur);
}
```

Nous récupérons l'identité de l'utilisateur courant et récupérons ses informations de groupe en passant `IdentiteUtilisateur` au constructeur de `WindowsPrincipal`.

Il existe une autre manière de créer notre objet `WindowsPrincipal`, sans utiliser `WindowsIdentity`. Vous pouvez directement extraire un objet `WindowsPrincipal` du thread courant. En effet le thread courant possède un contexte d'exécution et possède donc les informations nécessaires à la création d'un objet `WindowsPrincipal` pour l'utilisateur courant.

Tout d'abord vous devrez configurer les paramètres de sécurité du thread afin qu'ils utilisent les paramètres de sécurité de Windows plutôt que ceux utilisés par défaut au démarrage de l'application. Ensuite vous devrez caster en `WindowsPrincipal` l'objet `Thread.CurrentPrincipal`.

```
'VB
Imports System.Threading
Sub Main()
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsP
rincipal)
    Dim GroupeUtilisateur As WindowsPrincipal =
    CType(Thread.CurrentPrincipal, WindowsPrincipal)
End Sub

//C#
using System.Threading;
static void Main()
{
    AppDomain.CurrentDomain.SetPrincipalPolicy(
    PrincipalPolicy.WindowsPrincipal);
    WindowsPrincipal GroupeUtilisateur =
    (WindowsPrincipal)Thread.CurrentPrincipal;
}
```

La classe `WindowsPrincipal` contient deux membres intéressants :

Membres	Description
<code>Identity</code>	Retourne un objet <code>WindowsIdentity</code> avec l'identité de l'utilisateur.
<code>IsInRole</code>	Retourne si l'utilisateur fait partie ou non du groupe passé en paramètre.

Dans le cas de `WindowsPrincipal`, les rôles sont les groupes utilisateurs du système Windows.

Nous allons voir grâce à un exemple trois façons d'utiliser la méthode `IsInRole` :

- La première méthode consiste à passer une valeur de l'énumération `WindowsBuiltInRole` en tant que paramètre
- La seconde méthode consiste à passer sous forme de chaîne de caractère le nom du groupe
- La troisième méthode consiste à passer sous forme de chaîne de caractère le nom du domaine suivi d'un antislash \ et du nom du groupe.

```
'VB
Sub Main()
    Dim IdentiteUtilisateur As WindowsIdentity =
WindowsIdentity.GetCurrent()
    Dim GroupeUtilisateur = New WindowsPrincipal(IdentiteUtilisateur)
    If (GroupeUtilisateur.IsInRole(WindowsBuiltInRole.Administrator))
Then
    Console.WriteLine(IdentiteUtilisateur.Name + " fait partie
des Administrateurs")
    End If
    If (GroupeUtilisateur.IsInRole("Utilisateurs")) Then
        Console.WriteLine(IdentiteUtilisateur.Name + " fait partie
des Utilisateurs locaux")
    End If
    If (GroupeUtilisateur.IsInRole("TEST\Utilisateurs")) Then
        Console.WriteLine(IdentiteUtilisateur.Name + " fait partie
des Utilisateurs du domaine TEST")
    End If
End Sub

//C#
static void Main()
{
    WindowsIdentity IdentiteUtilisateur = WindowsIdentity.GetCurrent();
    WindowsPrincipal GroupeUtilisateur = new
WindowsPrincipal(IdentiteUtilisateur);
    if (GroupeUtilisateur.IsInRole(WindowsBuiltInRole.Administrator))
        Console.WriteLine(IdentiteUtilisateur.Name + " fait partie des
Administrateurs");
    if (GroupeUtilisateur.IsInRole(@"Utilisateurs"))
        Console.WriteLine(IdentiteUtilisateur.Name + " fait partie des
Utilisateurs locaux");
    if (GroupeUtilisateur.IsInRole(@"TEST\Utilisateurs"))
        Console.WriteLine(IdentiteUtilisateur.Name + " fait partie des
Utilisateurs du domaine TEST");
    Console.Read();
}
```

SCHMAC\Paul fait partie des Administrateurs

SCHMAC\Paul fait partie des Utilisateurs locaux

Nous avons utilisé successivement les trois méthodes dans cet exemple. Dans la première nous vérifions si l'utilisateur fait partie du groupe des administrateurs, c'est le cas, la condition est donc bonne on affiche que l'utilisateur fait partie des Administrateurs.

Dans le second cas, on vérifie si l'utilisateur fait partie du groupe Utilisateurs, c'est le cas on l'affiche. Il n'est pas intéressant d'utiliser cette méthode pour vérifier une valeur contenue dans l'énumération `WindowsBuiltInRole`, vous utiliserez plutôt cette façon de faire pour vérifier l'appartenance de notre utilisateur à un groupe que vous avez créé.

Enfin dans le dernier cas, nous rajoutons un domaine - TEST- comme notre utilisateur ne fais pas partie du domaine TEST, la condition n'est pas vérifiée et rien n'est affiché.



#### 4.1.4 Crédation de classe d'identité de groupe personnalisée

Tout comme `WindowsIdentity` hérite de `IIdentity`, `WindowsPrincipal` hérite de l'interface `IPrincipal`. Pour créer une classe d'identité de groupe personnalisée, vous devrez donc hériter et implémenter `IPrincipal`.

Votre classe devra obligatoirement comporter un constructeur qui va permettre de récupérer les informations d'identité d'un objet `IIdentity`, la méthode `IsInRole`, et la propriété `Identity`.

#### 4.1.5 Authentification générique

Avec `WindowsIdentity`, nous avons créée un objet représentant l'identité d'un utilisateur Windows. `WindowsPrincipal` représentait lui les groupes utilisateurs auquel l'utilisateur appartient. Il y a plusieurs autres types d'utilisateurs et de groupes d'utilisateurs que vous pouvez avoir besoin de représenter. Dans la plupart des cas, vous devrez créer vos propres classes implémentant `IIdentity` et `IPrincipal`. Le Framework .NET fournit malgré tout deux autres classes permettant de représenter utilisateur et groupe d'utilisateurs : `GenericIdentity` et `GenericPrincipal`.

Ces deux classes permettent de créer des utilisateurs virtuels afin de gérer facilement les droits d'une application simple. Voici un exemple de création des objets `GenericIdentity` et `GenericPrincipal` :

```
'VB
Sub Main()
    Dim utilisateur As GenericIdentity = New GenericIdentity("Toto",
"Biometrie")
    Dim roles() As String = New String() {"Utilisateurs",
"Administrateurs"}
    Dim GroupeGenerique As GenericPrincipal = New
GenericPrincipal(utilisateur, roles)
End Sub

//C#
static void Main()
{
    GenericIdentity utilisateur = new GenericIdentity("Toto",
"Biometrie");
    string[] roles = new string[] {"Utilisateurs", "Administrateurs"};
    GenericPrincipal GroupeGenerique = new GenericPrincipal(utilisateur,
roles);
}
```

D'abord nous créons l'objet `GenericIdentity`, nous mettons en paramètre son nom d'utilisateur, et la méthode d'authentification.

Ensuite nous instancions l'objet `GenericPrincipal`, nous passons en paramètre l'objet `GenericIdentity` précédent, et un tableau des rôles auquel il appartient, Users et Administrators.

Si vous voulez maintenant faire en sorte que les paramètres de sécurité du Thread soient associés à cet utilisateur générique, il suffit de placer votre identité générique dans la propriété `CurrentPrincipal` du thread :

```
'VB
Sub Main()
    Dim utilisateur As GenericIdentity = New GenericIdentity("Toto",
"Biometrie")
    Dim roles() As String = New String() {"Utilisateurs",
"Administrateurs"}
    Dim GroupeGenerique As GenericPrincipal = New
GenericPrincipal(utilisateur, roles)
    Thread.CurrentPrincipal = GroupeGenerique
    Console.WriteLine(Thread.CurrentPrincipal.Identity.Name)
    Console.Read()
End Sub

//C#
static void Main()
{
    GenericIdentity utilisateur = new GenericIdentity("Toto",
"Biometrie");
    string[] roles = new string[] { "Users", "Administrators" };
    GenericPrincipal GroupeGenerique = new GenericPrincipal(utilisateur,
roles);
    Thread.CurrentPrincipal = GroupeGenerique;
    Console.WriteLine(Thread.CurrentPrincipal.Identity.Name);
    Console.Read()
}
```

Toto

Nous associons notre objet `GenericPrincipal` à `Thread.CurrentPrincipal` et pour vérifier, on affiche `Identity.Name`.

## 4.2 Autorisation

Dans le processus de sécurisation des données, l'autorisation est une phase clé de l'accès aux données. En effet, sans elle, il vous sera impossible d'avoir accès à une partie des ressources.

Le .NET Framework vous donne la possibilité de vérifier que l'utilisateur qui exécute votre code possède bien le bon rôle (autrement dit, qu'il adhère au groupe d'utilisateur ayant les droits suffisants pour effectuer cette opération). Pour cela, vous utiliserez soit la classe PrincipalPermission (pour le test de façon impérative) soit l'attribut PrincipalPermission (pour le test de façon déclarative). Ces deux éléments se trouvent dans l'espace de nom System.Security.Permissions.

En ce qui concerne la méthode déclarative, vous pouvez spécifier les paramètres suivant :

- Name : Représente le nom complet de l'entité à vérifier. Par exemple, si nous souhaitons autoriser l'utilisateur "Jean" sur la machine "Pc-Familial", le nom complet serait "Pc-Familial\Jean" ("Pc-Familial" représente le domaine dans lequel se trouve la machine. Si vous vous authentifiez sur un réseau portant un nom de domaine, vous devrez spécifier le domaine suivi du login de l'utilisateur)
- Authenticated : Spécifie un booléen indiquant si l'utilisateur doit être authentifié ou non.
- Role : Indique le nom du groupe d'utilisateur dans lequel on doit se trouver pour correspondre aux critères de recherche.
- SecurityAction : Indique l'action de sécurité à entreprendre (demande, interdiction, ...)

Si vous préférez utiliser la façon impérative, les constructeurs de la classe PrincipalPermission vous permettent de spécifier :

- Name : Le nom d'utilisateur (comme pour la méthode déclarative)
- Role : Le nom du groupe d'utilisateur (comme pour la méthode déclarative)
- Authenticated : La restriction d'authentification (comme pour la méthode déclarative)

Ou alors l'un des valeurs de l'énumération **PermissionState**. Cette énumération vous fourni deux valeurs :

- **None** : Qui indique que l'utilisateur n'a aucun accès aux ressources
- **Unrestricted** : Qui indique que l'utilisateur a un accès total aux ressources.

Dans le code suivant, nous avons créé une méthode qui a besoin qu'un utilisateur authentifié et ayant les droits d'administration exécute l'application. Si l'utilisateur courant ne respecte pas l'une de ces conditions, une exception sera levée :

```
'VB
Imports System.Security.Permissions

Sub Main()
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal)

    Try
        Admin()
    Catch ex As Exception
        Console.WriteLine("Autorisation refusée" + vbCrLf +
ex.Message)
    End Try
    Console.Read()
End Sub

<PrincipalPermission(SecurityAction.Demand, Authenticated:=True,
Role:="Administrateurs")> _
Public Sub Admin()
    Console.WriteLine("Autorisation accordée!")
End Sub

//C#
using System.Security.Permissions;

static void Main()
{
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
    try
    {
        Admin();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Autorisation refusée\n" + ex.Message);
    }
    Console.Read();
}
[PrincipalPermission(SecurityAction.Demand, Authenticated=true,
Role="Administrateurs")]
public static void Admin()
{
    Console.WriteLine("Autorisation accordée!");
}
```

Si l'utilisateur authentifié actuellement fait partie du groupe "Administrateurs", nous aurons le résultat suivant :

Autorisation accordée!

Dans le cas contraire, l'exception sera levée et la méthode ne sera pas exécutée :

Autorisation refusée

Échec de la demande d'autorisation principale.



## 5 Gestion de la RBS Windows

Dans le [préambule](#) de ce chapitre, nous avions abordé la sécurité de la CLR en faisant une analogie avec le système de sécurité utilisé par Windows pour protéger l'accès aux ressources (fichiers, base de registre, connexions réseaux, ...) des utilisateurs non-autorisés.

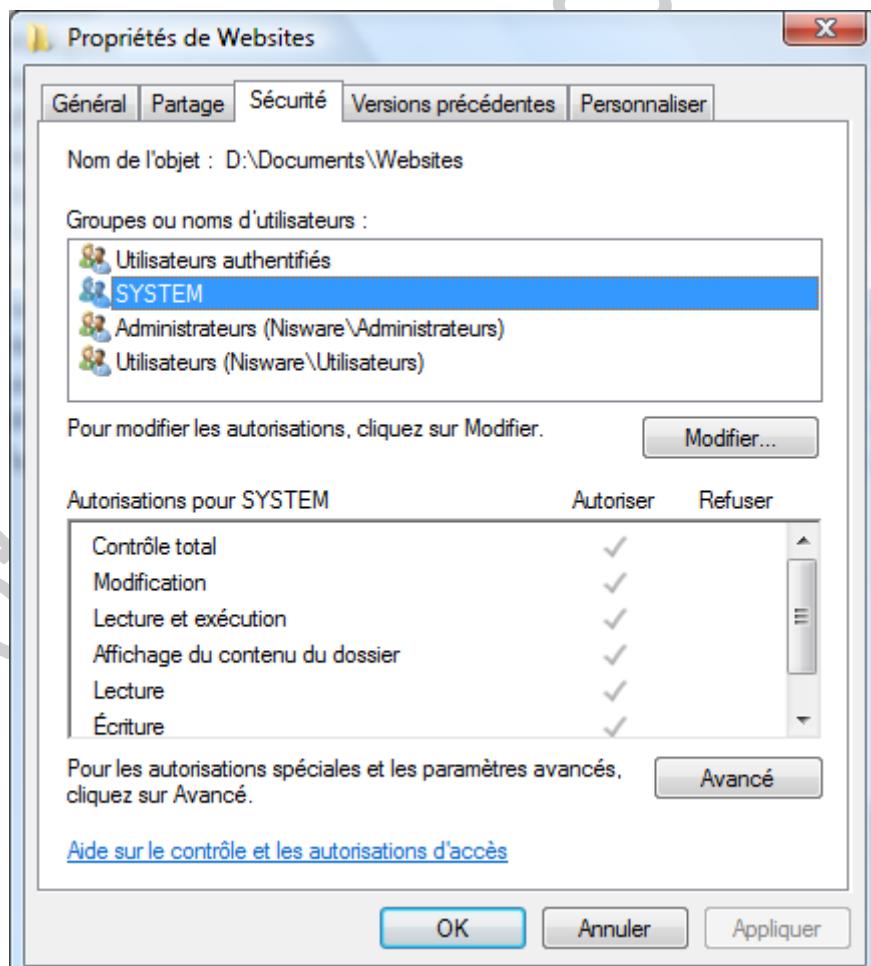
Dans cette partie, nous revenons sur le système RBS afin de voir comment le Framework peut agir sur ce système afin de régler les paramètres de sécurité.

### 5.1 Rappel sur le système RBS Windows

En entreprise, il n'est pas rare que des employés doivent se servir de l'outil informatique. Seulement, sans système de protection, un employé peu qualifié en ce qui concerne l'utilisation d'un ordinateur peut rapidement devenir un désavantage considérable pour l'entreprise (Imaginez si un employé d'une banque venait à supprimer par erreur toutes les transactions bancaires réalisées au cours des dernières 24 heures...).

Pour cela, les machines équipées d'un système Windows utiliseront un système d'utilisateurs et de groupe d'utilisateurs ; Chacun d'entre eux possédant des droits (ou rôle) bien défini.

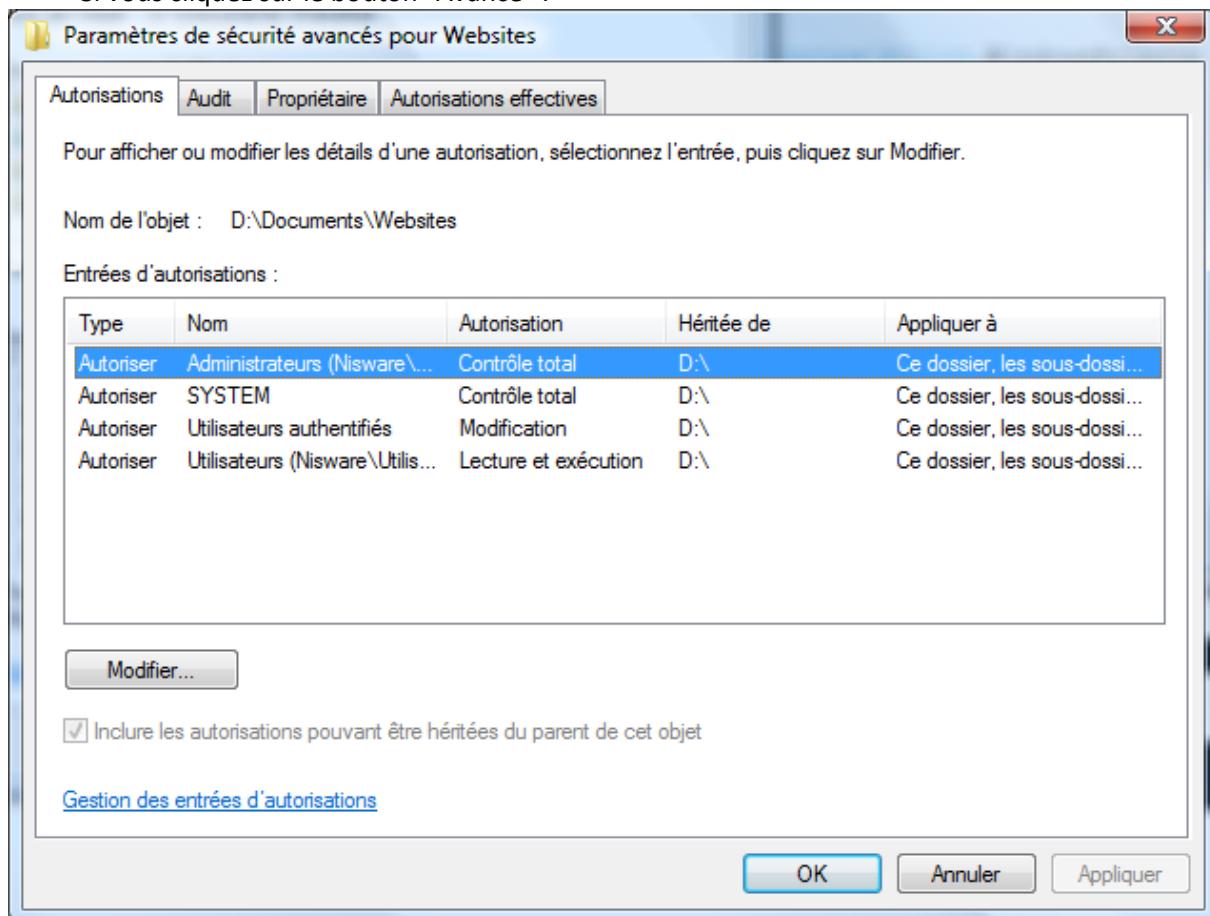
Si vous faites clic-droit sur un fichier, un dossier ou un disque dur, que vous faites propriétés puis que vous saisissez l'onglet "Sécurité", vous aurez accès à la gestion des droits sur l'élément concerné (Attention : Sur certaines machines, notamment celles équipées de versions dites "familiale" de Windows, vous devez configurer l'affichage des propriétés pour avoir accès à la sécurité) :



Dans le premier encadré, vous avez la liste des entités dont les droits sont modifiés (acceptés ou refusés) sur l'élément et dans l'encadré du dessous, vous avez la liste des droits effectifs appliqués à l'entité sélectionnée au dessus.



Si vous cliquez sur le bouton "Avancé" :



Vous aurez accès aux informations précises concernant la sécurité de l'élément :

- Autorisations : Cet onglet vous permet de visualiser ou modifier les droits effectifs sur les entités indiquées. Il vous permettra également de savoir si une permission est héritée de son conteneur parent ou non. Si la permission est effectivement héritée, vous devrez soit modifier les permissions du conteneur parent ou bien désactiver l'héritage des permissions sur l'élément courant (case grisée sur ce Screenshot. Il faut cliquer sur "Modifier" pour changer cette option). Toute cette section vous permettra de régler la liste de contrôle d'accès discrétionnaire (DACL pour Discretionary Access Control List)
- Audit : Il va vous permettre de gérer l'audit de l'élément en cours. Un audit, c'est le fait de surveiller toute action réalisée sur la ressource concernée, que ça soit la lecture, la modification, la suppression, le partage, ... . Vous pourrez gérer ce qu'on appelle les listes de contrôle d'accès de sécurité (SACL pour Security Access Control List). Retenez bien que les SACL ne règlent aucune restriction d'accès. Elles se contentent juste de paramétriser la surveillance (l'audit) de la ressource!
- Autorisations effectives : Cet onglet vous permet de tester quels sont les droits dont dispose un utilisateur ou un groupe sur la ressource en cours. Il vous suffit de saisir l'entité que vous souhaitez tester et de valider pour voir s'afficher les droits dont elle dispose ; Très pratique lorsque vous configurez des règles complexes en utilisant plusieurs groupes différents.

En regardant un peu ces droits d'accès, vous aurez rapidement constaté que c'est donc ici que l'on paramètrera qui peut avoir accès en lecture, en écriture... .



## 5.2 Gestion des droits via le code

S'il est possible de modifier les droits d'accès aux ressources en passant par le système d'exploitation, il est également possible de les modifier en utilisant les outils de l'espace de nom `System.Security.AccessControl`.

Dans cet espace de noms, vous trouverez tous les outils nécessaires à la vérification et à la modification des droits d'accès aux ressources. De façon générale, vous trouverez les types de classes suivants :

- <Ressources>Security (par exemple `RegistrySecurity`) qui vous permettra de visualiser/modifier aussi bien les listes DACL que les listes SACL.
- <Ressources>AccessRule (par exemple `RegistryAccessRule`) qui vous permettra de visualiser/modifier les DACL.
- <Ressources>AuditRule (par exemple `RegistryAuditRule`) qui vous permettra de visualiser/modifier les SACL.

Par défaut, le Framework implémente la classe `NativeObjectSecurity` pour différentes ressources :

- `FileSystemSecurity`, pour le système de fichier.
- `FileSecurity`, qui est spécifique aux fichiers
- `RegistrySecurity`, permettant la gestion de l'accès au registre Windows
- `MutexSecurity`, pour la sécurité inter-application des Mutex (voir chapitre 7)
- `SemaphoreSecurity`, identique à `MutexSecurity` mais pour les Sémaphores
- `EventWaitHandleSecurity`, pour la gestion des WaitHandle (voir chapitre 7)
- `DirectorySecurity`, qui est spécifique aux dossiers.

Grâce à l'une de ces classes, vous allez pouvoir altérer les droits d'accès à l'un des éléments cités ci-dessus.

Comme exemple, nous allons lister les droits appliqués au fichier C:\test.txt puis nous allons ajouter le droit de suppression et de modification au groupe d'utilisateurs "Utilisateurs" :

```
'VB
Imports System.Security.AccessControl
Imports System.Security.Principal
Imports System.IO

Sub Main()
    Dim fichier As String = "C:\test.txt"
    Dim secu As FileSecurity = New FileSecurity(fichier,
AccessControlSections.All)

    Dim dacl As AuthorizationRuleCollection = secu.GetAccessRules(True,
True, GetType(NTAccount))
    For Each ar As FileSystemAccessRule In dacl
        Console.WriteLine(ar.IdentityReference.Value + " a le(s)
droit(s) " + ar.FileSystemRights.ToString())
    Next

    Dim ndacl As FileSystemAccessRule = New
FileSystemAccessRule("Utilisateurs", FileSystemRights.Delete Or
FileSystemRights.Modify, AccessControlType.Allow)
    secu.AddAccessRule(ndacl)
    File.SetAccessControl(fichier, secu)
    Console.Read()
End Sub
```



```
// C#
using System.Security.AccessControl;
using System.Security.Principal;
using System.IO;
static void Main(string[] args)
{
    string fichier = @"C:\test.txt";
    FileSecurity secu = new FileSecurity(fichier,
    AccessControlSections.All);

    AuthorizationRuleCollection dacl =
secu.GetAccessRules(true,true,typeof(NTAccount));
    foreach(FileSystemAccessRule ar in dacl)
        Console.WriteLine(ar.IdentityReference.Value + " a le(s) droit(s)
" + ar.FileSystemRights.ToString());

    FileSystemAccessRule ndacl = new FileSystemAccessRule("Utilisateurs",
FileSystemRights.Delete | FileSystemRights.Modify,
AccessControlType.Allow);

    secu.AddAccessRule(ndacl);
    File.SetAccessControl(fichier, secu);
    Console.Read();
}
```

Dans la console, vous devriez avoir un résultat similaire à celui-ci :

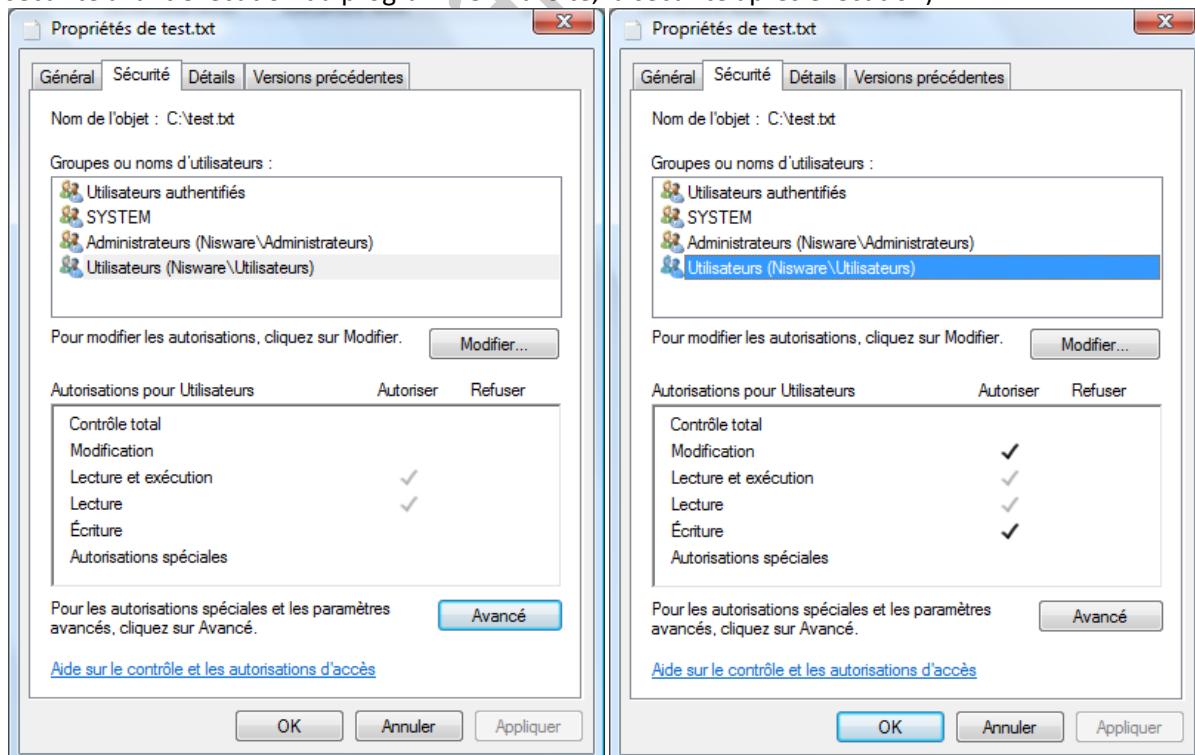
AUTORITE NT\Utilisateurs authentifiés a le(s) droit(s) Modify, Synchronize

AUTORITE NT\SYSTEM a le(s) droit(s) FullControl

BUILTIN\Administrateurs a le(s) droit(s) FullControl

BUILTIN\Utilisateurs a le(s) droit(s) ReadAndExecute, Synchronize

Et si vous allez regarder la sécurité du fichier, vous devriez obtenir ce changement (à gauche, la sécurité avant exécution du programme. A droite, la sécurité après exécution) :





Dans l'exemple, vous aurez pu constater que lorsqu'on demande à récupérer la liste DACL ou SACL, trois paramètres sont nécessaires. Les deux premiers indiquent respectivement si la recherche doit inclure les droits spécifiques à l'élément et les droits hérités du conteneur parent. Le troisième paramètre est un objet de type Type représentant le "formatage" utilisé pour représenter les entités. Vous ne pouvez pas y passer n'importe quel type! Vous ne pourrez utiliser que :

- Le type `System.Security.Principal.NTAccess`, pour obtenir une représentation sous forme de texte des entités.
- Le type `System.Security.Principal.SecurityIdentifier`, permettant d'obtenir une représentation en utilisant les identifiants numériques des entités (SID).



**Important :** Pour pouvoir modifier les droits d'accès à une ressource, vous devez, que ça soit via le système d'exploitation ou via le code, posséder le droit de modification des règles de sécurité. Par exemple, sous Windows Vista, vous ne pourrez pas modifier les permissions sur C:\Windows tant que vous êtes un membre du groupe Utilisateurs ou Administrateurs!



## 6 La cryptographie dans le .NET

Sécuriser les données au niveau de l'application est déjà une très bonne chose. Cependant, il est tout aussi important de brouiller les données importantes qui seront stockées ou échangées via un réseau. Aussi, pour cela, le .NET Framework implémente à peu près toutes les techniques de cryptographie traditionnelle. Cela inclus les méthodes de cryptage/décryptage de données, la signature de celles-ci ou encore le contrôle d'intégrité.

### 6.1 Empreintes des données

On appelle empreinte de données (ou code de hachage) toutes combinaisons d'octets qui, calculée à partir d'un algorithme particulier, identifie de façon unique des données. Ils permettent de vérifier l'intégrité des données. Ces codes de hachage ont toujours une taille de sortie fixe (plus ou moins grande en fonction de l'algorithme utilisé). Elle peut donc être supérieure à la taille des données dont on veut l'empreinte.

Dans les méthodes de calcul des empreintes de hachage, on distingue deux types d'algorithme :

- Les algorithmes sans clé, c'est-à-dire qu'ils ne prennent en entrée que les données dont on souhaite l'empreinte et retournent une série d'octets représentant l'empreinte numérique des données.

- Les algorithmes à clé ou algorithmes [HMAC](#). Pour calculer une empreinte des données, ces algorithmes nécessitent une clé secrète en plus des données afin de protéger l'empreinte générée. Ils sont en général un ajout aux algorithmes sans clés (Par exemple HMAC-MD5, version avec clé de l'empreinte MD5) et de ce fait sont utilisables avec n'importe quel algorithme de calcul d'empreinte (à partir du moment où cette méthode utilise une itération dans le calcul de l'empreinte). Ils sont utilisés dans le cas où les personnes échangeant des données peuvent également échanger la clé secrète utilisée pour générer le code de hachage.

Le Framework .NET fournit un ensemble de classes héritant de la classe [HashAlgorithm](#) pour les algorithmes de hachage sans clé ou de la classe [KeyedHashAlgorithm](#) pour les algorithmes de hachage avec clé qui permettent de calculer l'empreinte numérique de données :

Classe	Description
<a href="#">MD5CryptoServiceProvider</a>	Hachage MD5 générant des empreintes de 128bits
<a href="#">SHA1Managed</a>	Hachage SHA générant des empreintes de 160bits
<a href="#">RIPEMD160Managed</a>	Hachage MD160 générant des empreintes de 160bits
<a href="#">SHA256Managed</a>	Hachage SHA générant des empreintes de 256bits
<a href="#">SHA384Managed</a>	Hachage SHA générant des empreintes de 384bits
<a href="#">SHA512Managed</a>	Hachage SHA générant des empreintes de 512bits
<a href="#">MACTripleDES</a>	Hachage à clé secrète en utilisant l'algorithme 3DES.
<a href="#">HMACMD5</a> , <a href="#">HMACSHA1</a> ...	Hachage à clé secrète en utilisant l'un des algorithmes MD5, SHA1...

Toutes les classes de génération d'empreintes basées sur l'algorithme SHA disposent de deux versions : Une version entièrement managée (classes [SHA<version>Managed](#)) et une version d'adaptation des algorithmes implantés dans le service de chiffrement Windows (classes [SHA<version>CryptoServiceProvider](#)).

Dans l'exemple ci-dessous, nous calculons une empreinte SHA256 protégée avec une clé et une autre non protégée.

```
'VB
Imports System.Security.Cryptography
Imports System.Text
Imports System.IO
Sub Main()
    Dim rfc As Rfc2898DeriveBytes = New Rfc2898DeriveBytes("Mot de passe", 12)

    Dim hsha256 As HMACSHA256 = New HMACSHA256(rfc.GetBytes(64))
    Dim sha256 As SHA256Managed = New SHA256Managed()

    Dim b() As Byte = Encoding.Unicode.GetBytes("Phrase Hachée")
    hsha256.ComputeHash(b)
    sha256.ComputeHash(b)

    Console.WriteLine(hsha256.HashName + " génère l'empreinte " +
Convert.ToString(hsha256.Hash) + " protégée par la clé " +
Convert.ToString(hsha256.Key))
    Console.WriteLine("SHA256 génère l'empreinte " +
Convert.ToString(sha256.Hash))
    Console.Read()
End Sub

//C#
using System.Security.Cryptography;
using System.Text;
using System.IO;
static void Main(string[] args)
{
    Rfc2898DeriveBytes rfc = new Rfc2898DeriveBytes("Mot de passe", 12);

    HMACSHA256 hsha256 = new HMACSHA256(rfc.GetBytes(64));
    SHA256Managed sha256 = new SHA256Managed();

    byte[] b = Encoding.Unicode.GetBytes("Phrase Hachée");
    hsha256.ComputeHash(b);
    sha256.ComputeHash(b);

    Console.WriteLine(hsha256.HashName + " génère l'empreinte " +
Convert.ToString(hsha256.Hash) + " protégée par la clé " +
Convert.ToString(hsha256.Key));
    Console.WriteLine("SHA256 génère l'empreinte " +
Convert.ToString(sha256.Hash));

    Console.Read();
}
```

Lors de la compilation, nous constatons que l'empreinte protégée par mot de passe est toujours différente de la précédente. Ceci s'explique par le fait que nous utilisons la classe Rfc2898DeriveBytes qui se charge de générer une clé pseudo-aléatoire en se basant sur la phrase passée en argument. L'empreinte calculée en utilisant SHA256 seul est toujours la même.

SHA256 génère l'empreinte tW5lj4Lr32ujBvZD1ezSI5yyg6p4R4E1aUcyrNyW2Pl= protégée par la clé bEB8lHiuBb+VpeD2hKZck3BV1Xe/ap+OhGqUPCo2kb00gGJhrfLmT2/BaiaMWqAhZprrg

9Rdtnc38ZalWmpDcw==

SHA256 génère l'empreinte qUF3tT1eV8ojg8l8H//ShhoAUEAPUVRj4e4UIIHnF5k=

Même s'il est très peu probable que plusieurs données différentes aient la même empreinte, il faut veiller à bien choisir l'algorithme de calcul en fonction du niveau de sécurité requis. Par exemple, l'empreinte MD5 commence à montrer des faiblesses. Pour des applications demandant un haut niveau de sécurité, préférez les algorithmes SHA256, 384...



## 6.2 Chiffrement symétrique

Le chiffrement par clé symétrique est une méthode de chiffrement simple que vous avez peut être eu l'occasion d'utiliser. Elle consiste à utiliser une clé, souvent une suite de caractères, afin de passer d'un texte clair à un texte dit chiffré (indescriptible sans traitement préalable) et vice versa.

L'avantage du chiffrement par clé symétrique est la rapidité et la robustesse des algorithmes de traitement qui permettent de chiffrer de grandes quantités de données rapidement et sans perte.

L'inconvénient de cette méthode est l'utilisation d'une seule clé secrète pour chiffrer et déchiffrer. En effet, si une personne mal intentionnée arrive à trouver ou récupérer la clé secrète, il pourra visionner les données en clair. D'autre part, toutes les personnes susceptibles de chiffrer et déchiffrer les données devront s'échanger la clé secrète en clair.

### 6.2.1 Outil annexe : La classe Rfc2898DeriveBytes

Sous ce nom étrange se cache une classe permettant la génération de clés secrètes pour les échanges de données chiffrées par clé symétrique.

Pour créer votre clé secrète, vous allez devoir fournir à la classe trois informations :

- Un mot de passe ou une phrase à transformer
- Un Salt donnant à la génération d'octets un caractère pseudo-aléatoire plus complexe à retrouver.
- Le nombre d'itération nécessaire pour la création de la clé. (facultatif)

La manière la plus simple d'utiliser Rfc2898DeriveBytes est de passer les deux données au constructeur de la classe, puis de récupérer ensuite la clé générée grâce à la méthode GetBytes.

Voici un exemple d'utilisation :

```
'VB
Imports System.Security.Cryptography
Imports System.Text
Sub Main()
    Dim chaine As String = "MotDePasse"
    Dim salt() As Byte = Encoding.ASCII.GetBytes("CeciEstUnSalt")
    Dim cle As Rfc2898DeriveBytes = New Rfc2898DeriveBytes(chaine,
salt)
    Dim AES As RijndaelManaged = New RijndaelManaged()
    AES.Key = cle.GetBytes(AES.KeySize / 8)
End Sub

//C#
using System.Security.Cryptography;
using System.Text;
public static void Main(string[] args)
{
    string chaine = "MotDePasse";
    byte[] salt = Encoding.ASCII.GetBytes("CeciEstUnSalt");
    Rfc2898DeriveBytes cle = new Rfc2898DeriveBytes(chaine, salt);

    RijndaelManaged AES = new RijndaelManaged();
    AES.Key = cle.GetBytes(AES.KeySize / 8);
}
```

Cet exemple paraît plus complexe qu'il ne l'est vraiment. Nous commençons par créer notre chaîne à transformer en clé et notre salt. La chaîne est un string et le salt un tableau d'octets.

Ensuite nous créons notre objet de type Rfc2898DeriveBytes en passant au constructeur notre chaîne et notre salt.

Enfin nous instancions un objet représentant un algorithme de chiffrement que nous verrons dans la partie suivante. Ce qu'il faut en retenir se trouve dans la dernière ligne : Nous récupérons

grâce à GetBytes la clé et l'associons à notre algorithme. Pour récupérer une clé de la bonne taille, nous passons en paramètre la taille désirée, ici AES.KeySize représente la taille de la clé que nécessite notre algorithme en bits, que nous divisons par 8 pour obtenir la taille en octets.

Dotnet:France Association





### 6.2.2 Le chiffrement symétrique en pratique

Afin d'effectuer des chiffrements par clé symétrique, le Framework .NET fournit quelques classes dérivant de `System.Security.Cryptography.SymmetricAlgorithm` et contenues dans l'espace de nom `System.Security.Cryptography`.

Nous allons d'abord voir en détail les différentes classes puis ensuite leurs membres communs :

Classes	Longueur de la clé	Description
<code>DES</code>	56 bits	Acronyme de Data Encryption Standard. L'algorithme DES permet de chiffrer et déchiffrer grâce à des clés très courtes, de ce fait il est vulnérable au cracking par brute force. Les algorithmes RC visent à le remplacer.
<code>RC2</code>	Variable	Le RC2 est un standard visant à remplacer DES. Il utilise des clés de tailles variables.
<code>RijndaelManaged</code>	De 128 bits à 256bits par pas de 32bits.	Mieux connu sous le nom de AES (Advanced Encryption Standard). La classe RijndaelManaged est la seule qui permet un chiffrement par clé symétrique en appelant seulement du code managé. Les autres classes font appel à du code non managé.
<code>TripleDES</code>	156 bits dont 112 bits pour le chiffrement/déchiffrement	La classe TripleDES permet d'utiliser l'algorithme 3DES qui applique trois fois l'algorithme DES afin d'améliorer sa résistance au cracking.

Si nous devions vous conseiller un de ces algorithmes selon un critère de sécurité, nous choisirions Rijndael tout simplement car il permet le chiffrement de données en utilisant des clés secrètes de 256 bits difficilement crackable par une seule personne. De plus il utilise exclusivement du code managé ce qui le rend plus sécurisé dans un environnement .NET.



Voici maintenant les principaux membres de ces classes qui sont tous hérités de `System.Security.Cryptography.SymmetricAlgorithm`:

Membres	Description
<code>Key</code>	Permet d'obtenir ou de définir la clé secrète. La clé est générée automatiquement si aucune n'est spécifiée.
<code>KeySize</code>	Permet d'obtenir ou de définir la taille de la clé secrète en bits. Vous n'aurez normalement pas besoin de redéfinir cette propriété car le runtime choisit toujours la taille ayant le meilleur rapport rapidité/sécurité.
<code>LegalKeySizes</code>	Représente un tableau des tailles des clés secrètes utilisables par l'algorithme.
<code>Mode</code>	Permet d'obtenir ou de définir le mode de fonctionnement de l'algorithme. Pour définir le mode de fonctionnement, vous pourrez utiliser les valeurs de l'énumération <code>CipherMode</code> .
<code>IV</code>	Permet de définir ou d'obtenir le vecteur d'initialisation du chiffrement/déchiffrement. Ce vecteur est un ensemble de d'octets utilisés pour réaliser le chiffrement du premier paquet d'octets à chiffrer. Voir <a href="#">cet article</a> pour plus de détails.
<code>CreateDecryptor</code>	Crée un objet permettant le déchiffrement de données. L'objet est de type <code>ICryptoTransform</code> .
<code>CreateEncryptor</code>	Crée un objet permettant le chiffrement de données. L'objet est de type <code>ICryptoTransform</code> .
<code>GenerateKey</code>	Génère une nouvelle clé secrète, vous pourrez appeler cette méthode si vous souhaitez changer votre clé secrète.
<code>GenerateIV</code>	Génère un nouveau vecteur d'initialisation.
<code>ValidateKeySize</code>	Détermine si la taille de la clé spécifiée est valide pour l'algorithme choisi.

Afin de pouvoir déchiffrer une donnée, vous devrez créer un objet possédant la même clé, le même mode de chiffrement et le même IV que pour le chiffrement.

Maintenant que nous avons vu quelques bases sur le chiffrement symétrique, nous allons pouvoir comprendre comment tout cela fonctionne grâce à un exemple.



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

Sub Main()
    '(1)
    Dim chaine As String = "MotDePasse"
    Dim salt() As Byte = Encoding.ASCII.GetBytes("CeciEstUnSalt")
    Dim cle As Rfc2898DeriveBytes = New Rfc2898DeriveBytes(chaine,
salt)
    Dim AES As RijndaelManaged = New RijndaelManaged()
    AES.Key = cle.GetBytes(AES.KeySize / 8)
    Dim phrase As String = "Cette phrase va être chiffrée puis
déchiffrée"
    Dim fichier As String = "C:\test.txt"

    '(2)
    Dim encryptor As ICryptoTransform = AES.CreateEncryptor(AES.Key,
AES.IV)
    '(3)
    Dim fichierSortant As FileStream = New FileStream(fichier,
FileMode.Create, FileAccess.Write)
    Dim chiffre As CryptoStream = New CryptoStream(fichierSortant,
encryptor, CryptoStreamMode.Write)
    Dim ecris As StreamWriter = New StreamWriter(chiffre)
    '(4)
    ecris.WriteLine(phrase)

    ecris.Close()
    '(5)
    Dim fichierEntrant As FileStream = New FileStream(fichier,
FileMode.Open, FileAccess.Read)
    Dim decryptor As ICryptoTransform = AES.CreateDecryptor(AES.Key,
AES.IV)
    Dim dechiffre As CryptoStream = New CryptoStream(fichierEntrant,
decryptor, CryptoStreamMode.Read)
    Dim lis As StreamReader = New StreamReader(dechiffre)
    '(6)
    Console.WriteLine(lis.ReadToEnd())

    lis.Close()
    Console.ReadLine()
End Sub
```





```
//C#
using System.Security.Cryptography;
using System.IO;
using System.Text;
public static void Main(string[] args)
{
    //(1)
    string chaine = "MotDePasse";
    byte[] salt = Encoding.ASCII.GetBytes("CeciEstUnSalt");
    Rfc2898DeriveBytes cle = new Rfc2898DeriveBytes(chaine, salt);
    RijndaelManaged AES = new RijndaelManaged();
    AES.Key = cle.GetBytes(AES.KeySize / 8);
    string phrase = "Cette phrase va être chiffrée puis déchiffrée";
    string fichier = @"C:\test.txt";
    //(2)
    ICryptoTransform encryptor = AES.CreateEncryptor(AES.Key, AES.IV);
    //(3)
    FileStream fichierSortant = new FileStream(fichier, FileMode.Create,
    FileAccess.Write);
    CryptoStream chiffre = new CryptoStream(fichierSortant, encryptor,
    CryptoStreamMode.Write);
    StreamWriter ecris = new StreamWriter(chiffre);
    //(4)
    ecris.WriteLine(phrase);

    ecris.Close();
    //(5)
    FileStream fichierEntrant = new FileStream(fichier, FileMode.Open,
    FileAccess.Read);
    ICryptoTransform decryptor = AES.CreateDecryptor(AES.Key, AES.IV);
    CryptoStream dechiffre = new CryptoStream(fichierEntrant, decryptor,
    CryptoStreamMode.Read);
    StreamReader lis = new StreamReader(dechiffre);
    //(6)
    Console.WriteLine(lis.ReadToEnd());

    lis.Close();
    Console.ReadLine();
}
```

Cette phrase va être chiffrée puis déchiffrée

Pour expliquer cet exemple avec plus de facilité, j'ai placé dans le code des points de repères commentés ci-dessous.

- 1) Nous reprenons à l'identique l'exemple de Rfc2898DeriveBytes
- 2) Ici nous créons un objet qui va nous permettre de chiffrer la phrase, on le récupère grâce à la méthode CreateEncryptor.
- 3) Nous créons ici trois flux, et ce afin de ne pas avoir à utiliser des tableaux d'octets comme buffer, un flux en écriture sur un fichier, un flux de chiffrement pointant sur le flux de fichier, et un flux StreamWriter qui va piloter le tout.
- 4) Nous écrivons la phrase grâce à la méthode Write de StreamWriter, les données sont chiffrées et écrites dans le fichier.
- 5) Ensuite nous effectuons l'opération inverse, nous créons trois flux assez ressemblants, un sur le fichier en lecture, un qui va se charger de déchiffrer en lisant le fichier et un StreamReader qui va piloter le tout.
- 6) Enfin nous affichons le contenu du fichier déchiffré grâce à la méthode ReadToEnd de StreamReader.



### 6.3 Chiffrement asymétrique

Le cryptage des données de façon asymétrique est plus long à effectuer que le cryptage symétrique mais peut fournir dans certains cas, une sécurité accrue. En effet, le cryptage asymétrique n'utilise plus une seule clé secrète mais deux clés, l'une étant publique et l'autre privée. La clé privée n'est jamais diffusée et est utilisée pour décrypter les données lorsqu'elles sont reçues. La clé publique quand à elle est utilisée pour crypter les données avant leur envoi. Cette dernière est la seule clé à pouvoir être distribuée librement.

Si on se place dans le cas où une personne PA veut recevoir des données cryptées de PB de façon sécurisée, la personne PA va d'abord générer une clé privée qu'elle conservera secrète et une clé publique qu'elle enverra à PB. PB va recevoir la clé publique, encrypter les données à envoyer en utilisant cette clé publique et envoyer les données cryptées. PA va ensuite recevoir les données cryptées et va pouvoir décrypter ces données grâce à la clé privée. En admettant que quelqu'un ait intercepté le message crypté et la clé publique, le message resterait indéchiffrable facilement vu que la clé publique ne peut servir qu'au cryptage des données.

Du fait de la lenteur du processus de cryptage/décryptage asymétrique, il n'est, en général, pas utilisé pour chiffrer des quantités de données importantes. Par exemple, les connexions sécurisées [SSL/TLS](#) utilisent le cryptage asymétrique pour échanger le vecteur d'initialisation et la clé de chiffrement utilisés par le cryptage symétrique utilisé pour chiffrer les données transmises.

La classe de base permettant d'implémenter une méthode de cryptage asymétrique est la classe abstraite [AsymmetricAlgorithm](#) qui comporte, en plus des propriétés [KeySize](#) et [LegalKeySizes](#) les propriétés suivantes :

Membres	Description
<a href="#">KeyExchangeAlgorithm</a>	Permet d'obtenir l'algorithme d'échange des clés utilisées.
<a href="#">SignatureAlgorithm</a>	Permet d'obtenir une référence vers la signature utilisée.

Le Framework propose deux classes qui héritent de [AsymmetricAlgorithm](#). Ces deux classes sont en fait des outils de gestion (ou wrapper) de code non-managé qui fournissent ces systèmes de cryptage :



### 6.3.1 RSACryptoServiceProvider

Cette classe donne accès au cryptage [RSA](#). Cette méthode de cryptage vous permettra d'utiliser des clés dont la taille peut aller de 384 bits à 16384 bits! En voici les principaux membres :

Membres	Description
<a href="#">Clear</a>	Permet de vider toutes ressources en mémoire utilisées par le RSACryptoServiceProvider.
<a href="#">Decrypt</a>	Permet de déchiffrer des données cryptées par une clé publique.
<a href="#">Encrypt</a>	Permet d'encrypter des données en utilisant la clé publique.
<a href="#">ExportParameters</a>	Exporte les paramètres de clés publiques et/ou privées dans une structure <a href="#">RSAParameters</a> .
<a href="#">FromXmlString</a>	Importe les paramètres de clés à partir d'une chaîne XML.
<a href="#">ImportParameters</a>	Importe les paramètres de clés à partir d'une structure <a href="#">RSAParameters</a> .
<a href="#">SignData</a>	Crée une signature numérique pour un fichier en deux étapes : d'abord en générant un hash et ensuite en générant une signature basée sur ce hash.
<a href="#">SignHash</a>	Permet de signer l'empreinte de hash spécifiée en utilisant la clé privée.
<a href="#">ToXmlString</a>	Exporte les paramètres de clés sous forme de chaîne XML.
<a href="#">VerifyData</a>	Vérifie une signature numérique en la comparant à celle produite par <a href="#">SignData</a> .
<a href="#">VerifyHash</a>	Vérifie une signature numérique basée sur le hash du fichier.
<a href="#">PersistKeyInCsp</a>	Indique si le fournisseur de cryptage RSA doit stocker la clé privée dans le service de chiffrement CryptoAPI*.
<a href="#">UseMachineKeyStore</a>	Propriété statique indiquant si le fournisseur de cryptage RSA doit stocker la clé privée dans le magasin de clés de l'ordinateur.
<a href="#">PublicOnly</a>	Contient un booléen indiquant si le service de cryptage RSA actuel ne contient qu'une clé publique.

\*Toute machine Windows est équipée d'un service de chiffrement qui permet de stocker les clés privées dans une section propre à l'utilisateur actuel de la base de registre en plus de fournir les méthodes de chiffrement standard. Vous aurez plus d'informations en visitant la section [CryptoAPI](#) de MSDN ou encore un article concernant [le stockage des clés du service de chiffrement](#).

Voici un exemple concret de l'utilisation du service de cryptage RSA. Cet exemple a été fait de telle sorte que l'on puisse facilement le transposer sur un dialogue sécurisé de données via un réseau par exemple.

Nous avons deux méthodes : FournisseurDeDonnees et Destinataire. FournisseurDeDonnees représente la machine que l'on souhaite contacter à distance. Elle prendra en paramètre la clé publique que le destinataire aura pris soin de générer avant d'envoyer sa requête.

La méthode Destinataire doit être appelée deux fois : Une première fois sans arguments ce qui permettra de générer la clé privée et la clé publique et une seconde fois avec la clé privée en argument afin de lancer le déchiffrement des données enregistrées dans le fichier C:\test.bin.

Afin d'éviter d'utiliser des stockages du service de chiffrement de façon inutile, nous allons utiliser le stockage des clés sous formes d'objets RSAParameters. Dans le cas où vous auriez besoin de sauvegarder les clés privées dans le service de chiffrement, il vous suffira de créer une instance de la classe [CspParameters](#) en oubliant pas de spécifier un nom de conteneur (via la propriété [KeyContainerName](#)), de passer votre objet dans le constructeur de la classe [RSACryptoServiceProvider](#) et de placer la propriété [PersistKeyInCsp](#) à True :



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

Dim taillecle As Integer
Dim chemin_fichier As String
Dim _clepublic, _cleprivee As RSAParameters

Sub Main()
    taillecle = 384
    chemin_fichier = "C:\test.bin"
    Destinataire()
    FournisseurDeDonnees(_clepublic)
    Destinataire(_cleprivee)
    Console.Read()
End Sub

'Méthode générant les clés privées et publiques
Public Sub Destinataire()
    Dim keygen As RSACryptoServiceProvider = New
RSACryptoServiceProvider(taillecle)
    _clepublic = keygen.ExportParameters(False)
    _cleprivee = keygen.ExportParameters(True)
    Console.WriteLine("Destinataire a générée les clés suivantes:" +
vbNewLine + "Cle privée: " + keygen.XmlString(True) + vbNewLine +
vbNewLine + "Cle publique: " + keygen.XmlString(False) + vbNewLine)
    keygen.Clear()
End Sub

'Méthode de décryptage des données à l'aide de la clé privée
Public Sub Destinataire(ByVal cleprivee As RSAParameters)
    Dim fichier As FileStream = New FileStream(chemin_fichier,
 FileMode.Open)
    Dim octets(fichier.Length - 1) As Byte
    fichier.Read(octets, 0, fichier.Length)
    fichier.Close()

    Console.WriteLine("Texte avant décryptage: {0}",
Encoding.Unicode.GetString(octets))

    Dim decrypt As RSACryptoServiceProvider = New
RSACryptoServiceProvider(taillecle)
    decrypt.ImportParameters(cleprivee)

    Dim finaltxt As StringBuilder = New StringBuilder()
    Dim o() As Byte = decrypt.Decrypt(octets, False)

    Console.WriteLine("Texte après décryptage: {0}",
Encoding.Unicode.GetString(o))
End Sub
```



```
'VB - Suite
'Méthode cryptant les données en utilisant la clé publique
Public Sub FournisseurDeDonnees(ByVal clepublic As RSAParameters)
    Dim encryptor As RSACryptoServiceProvider = New
RSACryptoServiceProvider(taillecle)
    encryptor.ImportParameters(clepublic)

    Console.WriteLine("Fournisseur de données." + vbNewLine + "Clé
privée disponible?" + (Not encryptor.PublicOnly).ToString() + vbNewLine)

    Dim fichier As FileStream = New FileStream(chemin_fichier,
FileMode.Truncate)

    Dim o() As Byte =
encryptor.Encrypt(Encoding.Unicode.GetBytes("Donnees."), False)
    For Each b As Byte In o
        fichier.WriteByte(b)
    Next
    fichier.Flush()
    fichier.Close()
    encryptor.Clear()
End Sub
```

Dotnet-France ASSO



```

//C#
using System.Security.Cryptography
using System.IO
using System.Text

private static int taillecle;
private static string chemin_fichier;
private static RSAParameters _clepublic;
private static RSAParameters _cleprivee;
static void Main(string[] args)
{
    taillecle = 384;
    chemin_fichier = @"C:\test.bin";
    Destinataire();
    FournisseurDeDonnees(_clepublic);
    Destinataire(_cleprivee);

    Console.Read();
}

//Méthode générant les clés privées et publiques
public static void Destinataire()
{
    RSACryptoServiceProvider keygen = new
    RSACryptoServiceProvider(taillecle);
    _clepublic = keygen.ExportParameters(false);
    _cleprivee = keygen.ExportParameters(true);
    Console.WriteLine("Destinataire a généré les clés suivantes:\n" +
                      "Cle privée: " + keygen.ToXmlString(true) + "\n\n"
+
                      "Cle publique: " + keygen.ToXmlString(false) +
"\n");
    keygen.Clear();
}

//Méthode de décryptage des données à l'aide de la clé privée
public static void Destinataire(RSAParameters cleprivee)
{
    FileStream fichier = new FileStream(chemin_fichier, FileMode.Open);
    byte[] octets = new byte[fichier.Length];
    fichier.Read(octets, 0, (int)fichier.Length);
    fichier.Close();

    Console.WriteLine("Texte avant décryptage: {0}",
Encoding.Unicode.GetString(octets));

    RSACryptoServiceProvider decrypt = new
    RSACryptoServiceProvider(taillecle);
    decrypt.ImportParameters(cleprivee);

    StringBuilder finaltxt = new StringBuilder();
    byte[] o = decrypt.Decrypt(octets, false);
    Console.WriteLine("Texte après décryptage: {0}",
Encoding.Unicode.GetString(o));
}

```



```

//C# - Suite
//Méthode cryptant les données en utilisant la clé publique
public static void FournisseurDeDonnees(RSAParameters clepublic)
{
    //Créé l'outil d'encryption des données
    RSACryptoServiceProvider encryptor = new
    RSACryptoServiceProvider(taillecle);
    encryptor.ImportParameters(clepublic);

    Console.WriteLine("Fournisseur de données.\n" + "Cle privée
disponible?" + (!encryptor.PublicOnly).ToString() + "\n");

    //Ecrit les données cryptées dans le fichier
    FileStream fichier = new FileStream(chemin_fichier,
    FileMode.Truncate);

    byte[] o = encryptor.Encrypt(Encoding.Unicode.GetBytes("Donnees."),
    false);
    foreach(byte b in o)
        fichier.WriteByte(b);

    fichier.Flush();
    fichier.Close();
    encryptor.Clear();
}
  
```

Pour exécuter correctement ce programme, vous devrez d'abord créer le fichier C:\test.bin.

Lorsque vous exécutez ce programme, vous devriez obtenir quelque chose de similaire à ça dans la console :

Destinataire a générée les clés suivantes:

Cle privée: <RSAKeyValue><Modulus>0H4COsdum97UQNSM0cZehGDaMrXA7rMYb3RhJGdDmvKHID
NNpnSEOtULcHDmN8AF</Modulus><Exponent>AQAB</Exponent><P>7+me3vc7g479qh6bhEMpdpt
IDOKGCqj</P><Q>3nkFX8qU2EDrZPlk3PDXIkJXdoopR303</Q><DP>AJjBJ2bQBBS/9sNauap9GOkkG
qFIPCUn</DP><DQ>z837Vx7DKbx6JDCKfIX4nere97rWFBDB</DQ><InverseQ>GaotrImHs4dXSjz0A
1/V/yKTYT1JnqYn</InverseQ><D>m7vT/IxlaAaM8xQRT8xl42nY8pHy2BnrqEpwl/T63ckE2vW9ImJ
rMQ0naN7WWTXN</D></RSAKeyValue>

Cle publique: <RSAKeyValue><Modulus>0H4COsdum97UQNSM0cZehGDaMrXA7rMYb3RhJGdDmvKH
IDNNpnSEOtULcHDmN8AF</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>

Fournisseur de données.

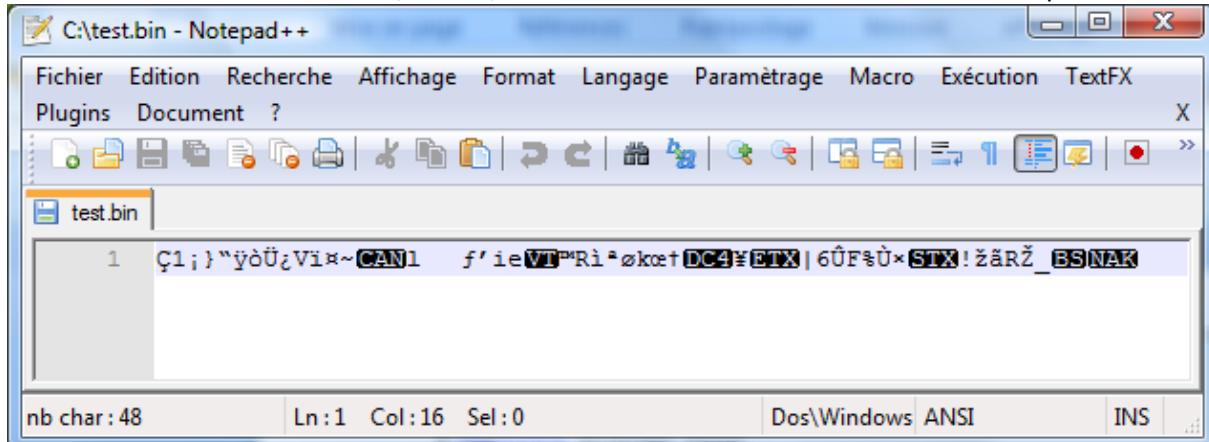
Cle privée disponible?False

Texte avant décryptage: ??????????????????????

Texte après décryptage: Donnees.



Si vous ouvrez le fichier C:\test.bin, vous devriez avoir un texte totalement incompréhensible :



Afin de déterminer quelle(s) clé(s) sera exportée(s) lors de l'appel à une méthode d'exportation, vous pouvez en général passer en argument un booléen indiquant :

- True : la clé privée sera exportée en plus de la clé publique.
- False : Seule la clé publique sera exportée.

Lorsque vous appelez une des méthodes de cryptage ou de décryptage, vous devez spécifier deux paramètres :

- Un tableau d'octets contenant les données à crypter.
- Un booléen indiquant si le service de chiffrement utilisera PKCS#1 v1.5 (essentiellement pour des soucis de rétrocompatibilités puisque disponible depuis plus longtemps, il est déconseillé de nos jours car il présente de nombreuses faiblesses) ou OAEP (disponible uniquement sur des machines XP ou plus récentes).

Vous aurez constaté que dans la clé privée au format XML, plusieurs balises sont créées. Chacune de ces balises représentent une partie des données permettant de générer les clés utilisées (pour savoir de quelle façon sont générées les clés, je vous renvoi vers le lien concernant l'algorithme [RSA](#). Sachez juste éventuellement que "Modulus" est le module de chiffrement n et "Exponent" est l'exposant de chiffrement e).

### 6.3.2 DSACryptoServiceProvider

Cette classe vous donne accès aux algorithmes de calcul [DSA](#). Elle possède les mêmes membres que [RSACryptoServiceProvider](#) mais ne dispose pas des méthodes [Encrypt](#) et [Decrypt](#). Du fait de l'absence de ces deux méthodes, le cryptage DSA ne peut pas être utilisé directement pour crypter des données. En revanche, il peut être d'une grande utilité si vous souhaitez implémenter un service fournissant des signatures, des clés publiques/privées et/ou des empreintes digitales de données.

## 6.4 Signatures numériques

Les signatures numériques permettent d'apposer à un document numérique une signature prouvant que le document vient bien de vous. Cela permet de vérifier la provenance d'un document et ainsi d'être sûr d'en détenir la bonne copie. Les signatures peuvent être vérifiées grâce à une clé publique fournie avec le document.

Les deux classes de chiffrements asymétriques vous permettent de signer les données grâce à leurs méthodes `SignData`, `SignHash`, `VerifyData` et `VerifyHash`.

Vous allez devoir, dans vos codes, utiliser conjointement `SignData` avec `VerifyData` ou `SignHash` avec `VerifyHash`. Nous allons illustrer l'utilisation des méthodes `SignData` et `VerifyData` de la classe `RSACryptoServiceProvider` en guise d'exemple. Sachez que l'utilisation de `DSACryptoServiceProvider` est quasi-identique, de même que l'utilisation des deux autres méthodes.





```
' VB
Imports System.Security.Cryptography
Imports System.IO
Sub Main()
    '(1)
    Dim RSA As RSACryptoServiceProvider = New
    RSACryptoServiceProvider()
    Dim clepublique As String = ""
    Dim signatureNumerique() As Byte
    '(2)
    signatureNumerique = Signature(RSA, clepublique)
    Console.WriteLine("La signature est : {0}",
    Convert.ToBase64String(signatureNumerique))
    '(3)
    Verification(RSA, signatureNumerique, clepublique)
    Console.Read()
End Sub

Public Function Signature(ByRef provider As RSACryptoServiceProvider,
ByRef clepublique As String) As Byte()
    '(4)
    Dim flux As FileStream = New FileStream("C:\test.txt",
    FileMode.Open, FileAccess.Read)
    Dim lecteur As BinaryReader = New BinaryReader(flux)
    '(5)
    Dim donnees() As Byte = lecteur.ReadBytes(CType(flux.Length,
    Integer))
    lecteur.Close()
    flux.Close()
    '(6)
    clepublique = provider.ToXmlString(False)
    '(7)
    Return provider.SignData(donnees, New
    SHA1CryptoServiceProvider())
End Function

Public Sub Verification(ByRef provider As RSACryptoServiceProvider, ByVal
signature() As Byte, ByRef clefpublique As String)
    '(8)
    provider.FromXmlString(clefpublique)
    Dim flux As FileStream = New FileStream("C:\test.txt",
    FileMode.Open, FileAccess.Read)
    Dim lecteur As BinaryReader = New BinaryReader(flux)
    '(9)
    Dim donnees() As Byte = lecteur.ReadBytes(CType(flux.Length,
    Integer))
    lecteur.Close()
    flux.Close()
    '(10)
    If (provider.VerifyData(donnees, New SHA1CryptoServiceProvider(),
    signature)) Then
        Console.WriteLine("La signature est vérifiée")
    Else
        Console.WriteLine("La signature est mauvaise")
    End If
End Sub
```



```
//C#
using System.Security.Cryptography;
using System.IO;
public static void Main(string[] args)
{
    //(1)
    RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
    string clepublique = "";
    byte[] signatureNumerique = null;
    //(2)
    signatureNumerique = Signature(ref RSA, ref clepublique);
    Console.WriteLine("La signature est : {0}",
    Convert.ToString(signatureNumerique));
    //(3)
    Verification(ref RSA, signatureNumerique, ref clepublique);
    Console.Read();
}
public static byte[] Signature(ref RSACryptoServiceProvider provider, ref
string clepublique)
{
    //(4)
    FileStream flux = new FileStream(@"C:\test.txt", FileMode.Open,
FileAccess.Read);
    BinaryReader lecteur = new BinaryReader(flux);
    //(5)
    byte[] donnees = lecteur.ReadBytes((int)flux.Length);
    lecteur.Close();
    flux.Close();
    //(6)
    clepublique = provider.ToXmlString(false);
    //(7)
    return provider.SignData(donnees, new SHA1CryptoServiceProvider());
}
public static void Verification(ref RSACryptoServiceProvider provider,
byte[] signature, ref string clefpublique)
{
    //(8)
    provider.FromXmlString(clefpublique);
    FileStream flux = new FileStream(@"C:\test.txt", FileMode.Open,
FileAccess.Read);
    BinaryReader lecteur = new BinaryReader(flux);
    //(9)
    byte[] donnees = lecteur.ReadBytes((int)flux.Length);
    lecteur.Close();
    flux.Close();
    //(10)
    if (provider.VerifyData(donnees, new SHA1CryptoServiceProvider(),
signature))
        Console.WriteLine("La signature est vérifiée");
    else
        Console.WriteLine("La signature est mauvaise");
}
```



Pour plus de clarté nous avons placé des numéros dans le code permettant de commenter chaque partie.

- 1) Nous instancions un objet RSACryptoServiceProvider qui va nous permettre de signer et vérifier la signature.
- 2) Nous appelons la méthode Signature, récupérons sa valeur de retour puis nous l'affichons. La méthode Signature signe un fichier retourne la signature et passe par référence la valeur de clepublique.
- 3) Nous appelons la méthode Verification qui va se charger de vérifier la signature du fichier.
- 4) On crée deux flux, un flux sur un fichier, et un BinaryReader utilisant le flux de fichier.
- 5) On lit entièrement le fichier et on remplit un tableau d'octets avec les informations recueillies
- 6) On récupère la clé publique sous forme XML (ici rangé dans une chaîne de caractère)
- 7) On signe le fichier à partir de son contenu (donnees) et avec un hash en SHA1, on retourne la signature.
- 8) On récupère la clé publique sous forme XML.
- 9) On lit le contenu du fichier et le range dans des tableaux d'octets (même étape que 5)
- 10) On vérifie la signature. A partir du contenu du fichier que l'on hash grâce à SHA1 on crée une nouvelle signature que l'on compare à celle passée en troisième paramètre.

Voici ce qui ressort dans la console. Bien entendu la signature est différente d'un fichier à un autre :

```
La signature est : LCgTxY3XYOPB3rTTLzT6dBas3x7lpwaaA2/HHaJNvcUgcHXe+EMhWKetUm9xTv37X1+wv5PUI7jh8KXwZ2POoHYFr3vhKFsvG31PcGCoyYKrjCkBJ9kDzdttdgAMo67xvAXI1Fxry1Vqh9VxyT7Xvw4GPfhS96u2R0/mc3ncsBs=
```

```
La signature est vérifiée
```

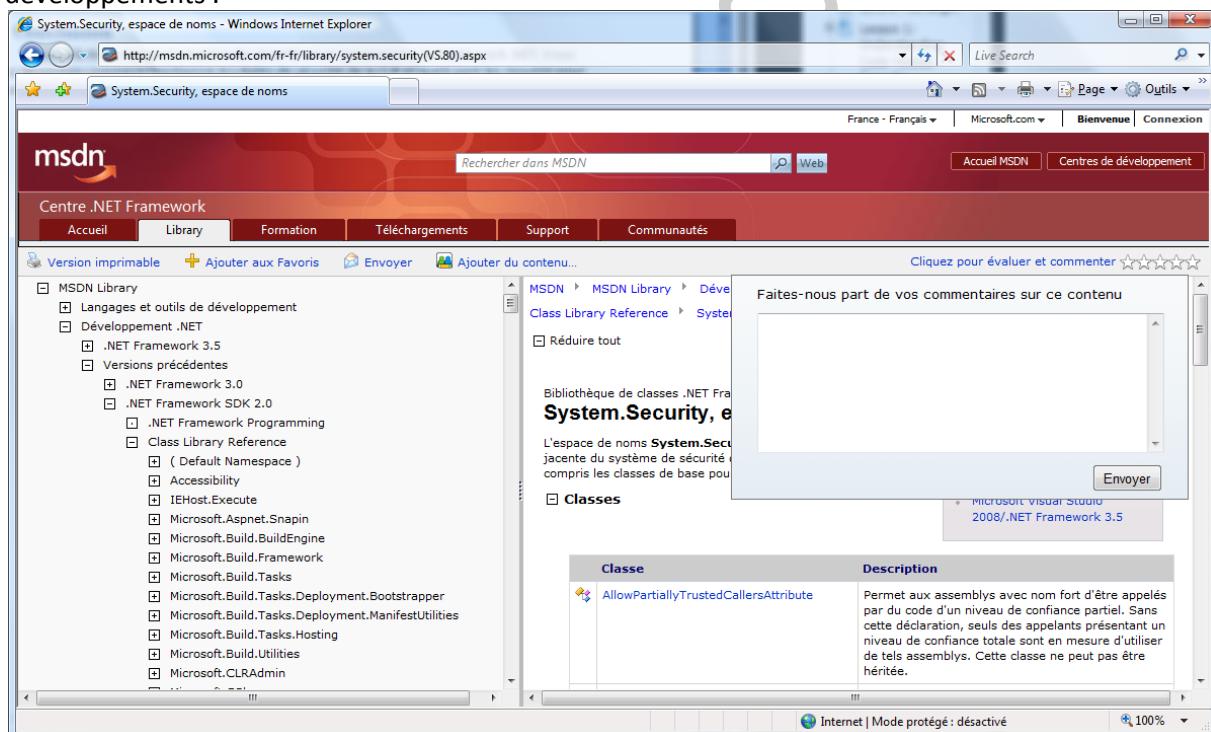
## 7 Conclusion

Vous voilà donc équipé en matière de sécurité en ce qui concerne le Framework .NET. Vous savez désormais comment fonctionne les règles de sécurité de la CLR et quels sont les moyens pour prévenir les exploits de vos applications ou encore le piratage de données sauvegardées.

A la fin de ce chapitre, vous devriez:

- Savoir comment fonctionne le CAS du Framework .NET et comment le configurer.
- Avoir compris comment fonctionnent les règles de sécurité par rapport au système d'exploitation hôte qui exécutera l'assembly.
- Savoir comment effectuer des demandes d'autorisations, comment en refuser que ça soit dans le code (au dessus des méthodes/classes) ou au niveau de l'assembly toute entière.
- Savoir comment gérer les ACLs (Discrétionnaire ou de sécurité) du système d'exploitation à partir du code.
- Connaitre les différentes techniques possibles pour chiffrer des données afin de les protéger lors de transfert ou pour les stocker à long terme sur un média (disque dur, cd-rom...)

N'oubliez pas que la [librairie](#) MSDN peut vous être d'une aide précieuse pour vos développements :



The screenshot shows a Microsoft Internet Explorer window displaying the MSDN Library. The URL in the address bar is [http://msdn.microsoft.com/fr-fr/library/system.security\(VS.80\).aspx](http://msdn.microsoft.com/fr-fr/library/system.security(VS.80).aspx). The page title is "System.Security, espace de noms - Windows Internet Explorer". The main content area shows the "System.Security" class library reference. On the left, there's a navigation tree for the MSDN Library, including sections like "Langages et outils de développement", ".NET Framework 3.5", "Versions précédentes", and "Class Library Reference". The right side of the page displays the class reference for "System.Security", specifically the "AllowPartiallyTrustedCallersAttribute" class. The class is described as allowing assemblies with a strong name to be called by code running at a lower trust level. The page also includes a feedback section for comments and a "Send Feedback" button.